UNIVERSITÀ DEGLI STUDI DI TRIESTE

Dipartimento di Ingegneria e Architettura

Laurea Triennale in Ingegneria Elettronica e Informatica

Sviluppo di un algoritmo di machine vision per applicazioni industriale

Laureando
Adriano Tumino

Relatore
Chiar.mo Prof. Sergio Carrato

Correlatore
Ing. Piergiorgio Menia

Anno Accademico 2018/2019

Indice

Introduzione

Nelle fonderie il metallo fuso viene riposto all'interno di alcuni recipienti e il loro contenuto identificato attraverso delle piastre di metallo, come in figura 1, saldate al contenitore stesso. Le piastre contengono un codice formato da cerchi ed inoltre degli angoli necessari per stabilirne l'orientamento corretto. In base alla posizione dei cerchi si può stabilire il codice del contenitore e relativo contenuto. Sarà quindi necessario esaminare un'immagine, trovare il riquadro di interesse che contiene il codice, per poi decodificarlo.
Questo è il principio alla base del progetto svolto presso la ELIMOS di Trieste. Lo scopo è quello di realizzare un algoritmo che permetta di prendere in input un'immagine, filtrarla fino ad ottenere la sola zona di interesse per poi passare alla sua decodifica. Questo algoritmo viene prima implementato nell'ambiente software MATLAB per poi essere trascritto in C++ con l'utilizzo della libreria OpenCV. Il lavoro è stato svolto interamente al calcolatore, realizzando test e simulazioni che riproducono situazioni reali. Durante questi collaudi vengono tenuti sotto controllo il tempo di esecuzione e l'utilizzo dei vari core della CPU disponibili.
Nel primo capitolo vengono forniti cenni teorici esaustivi sugli argomenti affrontati. In particolare, si illustrano i vari algoritmi, anche a livello matematico, per l'elaborazione delle immagini e i linguaggi di programmazione utilizzati sottolineando alcune funzioni importanti.
Nel secondo capitolo si evidenzia il ragionamento che sta dietro l'algoritmo implementato in MATLAB prestando particolare attenzione al passaggio dal caso ideale al caso reale e alle modifiche agli algoritmi stessi.
Nel terzo capitolo si affronta il passaggio da MATLAB a C++ avendo cura di descrivere le implementazioni di alcune funzioni di MATLAB non presenti in OpenCV.
Nel quarto capitolo si discute della fase di test effettuata.
Infine, vengono riportate le conclusioni che ricapitolano tutti i risultati ottenuti e di come successivamente il programma potrebbe essere ampliato.

Figura 1: Esempio di contenitore reale sul quale è applicata la matrice forata

Capitolo 1

Cenni Teorici

L'immagine digitale si può considerare come una funzione bidimensionale $f(x, y)$, dove x e y rappresentano le coordinate spaziali. Ogni punto dell'immagine prende il nome di pixel e questo possiede un valore discreto. L'immagine è limitata allo spettro elettromagnetico visibile (EM) [10] e se questa immagine è elaborata attraverso l'utilizzo di un calcolatore allora è possibile ricoprire quasi interamente tutto l'EM.
L'Elaborazione delle immagini digitali [11] è una materia che utilizza degli algoritmi di elaborazione numerica dei segnali, infatti essendo l'immagine una funzione bidimensionale allora è possibile trattarla come un segnale. Gli algoritmi restituiscono l'immagine di partenza modificata oppure delle informazioni di alcune caratteristiche dell'immagine stessa. Queste elaborazioni possono avvenire in maniera totalmente autonoma, effettuata interamente dal calcolatore, oppure in interazione continua con l'utente.
Si possono determinare tre livelli di elaborazioni:

- Basso livello [1.0.1];
- Medio livello [1.0.2];
- Alto livello [1.0.3];

1.0.1 Elaborazione di basso livello

L'elaborazione di basso livello prevede operazioni primitive, le quali servono per la preelaborazione delle immagini. Questo tipo di processo è caratterizzato dal fatto che il suo input ed il suo output sono entrambi delle immagini. Tra i principali esempi di questo tipo di processi troviamo la riduzione del rumore, il miglioramento del contrasto e della sua nitidezza.

1.0.2 Elaborazione di medio livello

L'elaborazione di medio livello è un processo i cui input generalmente sono immagini, ma i suoi output sono caratteristiche estratte dall'immagine. Tra

questi processi troviamo la ripartizione dell'immagine in regioni o oggetti, il riconoscimento degli oggetti e l'elaborazione di quest'ultimi.

1.0.3 Elaborazione di alto livello

Il livello superiore ha lo scopo di dare un significato ad un insieme di oggetti. Questo è detto analisi delle immagini e solitamente viene utilizzato per svolgere le funzioni cognitive associate alla vista.

1.1 Tipologia di Filtri

I filtri sono algoritmi che realizzano delle elaborazioni di segnali presenti al suo ingresso. Vi sono vari tipi di filtri che differiscono in prestazioni, accuratezza e complessità.
Qui consideriamo i filtri più importanti, trattandone gli aspetti teorici e matematici, utilizzati per il filtraggio dell'immagine durante l'algoritmo.

1.1.1 Edge Detection

Il riconoscimento dei contorni, in inglese edge detection, è un procedimento utilizzato per determinare i bruschi cambiamenti di intensità luminosa. Solitamente queste elevate variazioni sono dovuti a importanti alterazioni nella realtà, ad esempio il bordo di un oggetto. Questa operazione genera delle immagini contenenti poche informazioni, dato che vengono eliminati un gran numero di dati non rilevanti. Le informazioni rimanenti sono caratteristiche di tipo geometriche, che rappresentano degli oggetti. Esistono molti operatori per il riconoscimento dei contorni, tra questi troviamo gli algoritmi basati sulla ricerca tramite le derivate. Infatti, quest'ultimi riproducono i contorni attraverso il gradiente locale determinando i massimi e minimi locali.

Operatore di Sobel

L'operatore di Sobel è un operatore differenziale di due variabili, il quale calcola il gradiente locale recuperando alcune informazioni, ad esempio la direzione in cui si ha il massimo incremento chiaro-scuro e la velocità con cui avviene questo cambiamento. In particolare, nelle zone con la luminosità costante l'operatore di Sobel ha valore nullo, mentre nei punti situati su un contorno si ha un vettore con verso dal punto più scuro al più chiaro.
Essendo un operatore differenziale di sole due variabili si ha la necessità che l'immagine in input abbia solo un canale e quindi non è possibile applicare il filtro ad un immagine RGB [12].
Matematicamente questo operatore effettua la convoluzione di due matrici

con l'immagine in input. Queste due matrici, dette anche kernel, hanno dimensione 3×3 e vengono utilizzate come matrici di convoluzioni per il calcolo approssimato delle derivate parziali rispetto x e y.
Supponiamo sia I il nome dell'immagine in input allora le derivate nelle due direzioni sono date da:

$$G_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} * I \tag{1.1.1}$$

$$G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * I \tag{1.1.2}$$

dove $*$ indica il prodotto di convoluzione. È possibile calcolare il valore totale del gradiente come combinazione dei gradienti nelle due direzioni utilizzando la formula (1.1.3).

$$G = \sqrt{{G_x}^2 + {G_y}^2} \tag{1.1.3}$$

In particolare, la direzione è determinata attraverso la formula (1.1.4).

$$\theta = \arctan\left(\frac{G_y}{G_x}\right) \tag{1.1.4}$$

Questo operatore fornisce un valore approssimato del gradiente, ma la qualità è comunque sufficiente per molte operazioni successive.
Supponiamo di avere in input la figura 1.1 sulla quale si vuole applicare l'operatore di Sobel. Per far ciò vengono applicati i kernel all'immagine così da determinare sia le sue derivate parziali lungo le coordinate che il gradiente totale, attraverso la formula (1.1.3), ottenendo la figura 1.2.

Figura 1.1: Chiesa di San Giorgio, Ragusa Ibla

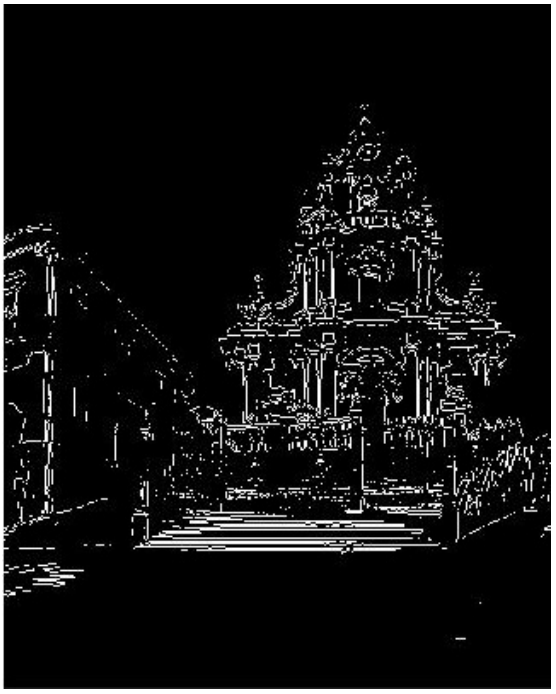

Figura 1.2: Esempio dell'operatore di Sobel applicata ad una foto: determina i bordi degli oggetti

Filtro di Canny

L'algoritmo di Canny, invece, utilizza un metodo di calcolo multi-stadio per individuare i contorni.
Per effettuare ciò, l'algoritmo di Canny utilizza un filtro costruito sulla derivata prima di una funzione gaussiana [13]. Spesso i risultati presentano dei disturbi dovuti al rumore contenuto nei dati di un'immagine, per questo motivo prima di applicare tale operatore l'immagine viene filtrata con un filtro gaussiano [2].
Per il calcolo del gradiente l'algoritmo utilizza quattro filtri differenti che hanno il compito di trovare i contorni orizzontali, verticali e diagonali dell'immagine. Per ogni pixel ottenuto se ne calcola la sua direzione e questa è data dal filtro, tra i quattro presenti, che assume il valore maggiore. La combinazione del valore e della direzione fornisce il massimo gradiente in ogni punto.
Vi sono molti punti che possono essere non di contorno, quindi si stabilisce che solo i massimi locali corrispondono ad un contorno, mentre i restanti vengono scartati.
Infine i contorni vengono estratti tramite un procedimento di thresholding [1.1.8] con isteresi [9]. Vengono definite due soglie, alta e bassa, le quali saranno confrontate con il gradiente. I punti accettati sono i valori che hanno valori maggiori della soglia alta, oppure che è compresa tra le due soglie, ma a condizione che un punto ad esso adiacente sia stato già accettato.
Supponiamo di avere in input la figura 1.1, allora applicando l'algoritmo di Canny si ottiene l'immagine 1.3.

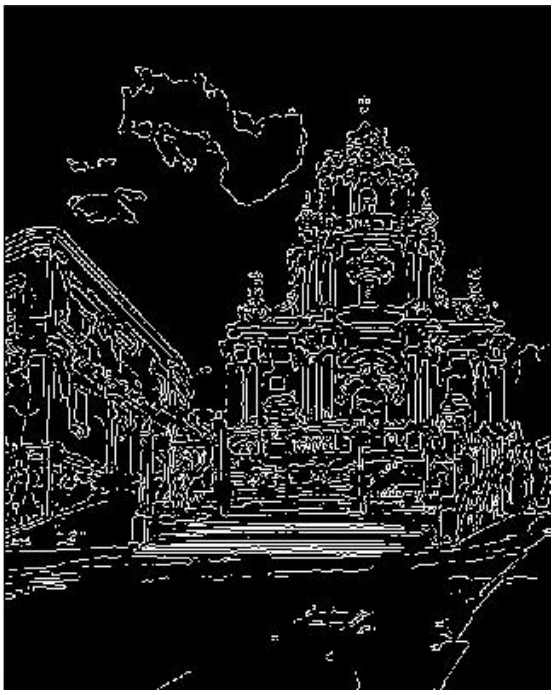

Figura 1.3: Esempio dell'algoritmo di Canny applicata ad una foto: Determina tutti i bordi degli oggetti con precisione

1.1.2 Erosione e Dilatazione

La dilatazione e l'erosione fanno parte dell'elaborazione morfologica matematica dell'immagine.
La dilatazione ha il compito di ampliare progressivamente i contorni degli oggetti, aumentandone le dimensioni e riducendo tutto il resto. Per effettuare la dilatazione si ha bisogno di un elemento strutturante [4].
Supponiamo sia I la matrice dell'immagine in input, B la matrice dell'elemento strutturante, allora per calcolare la dilatazione bisogna sovrapporre B ad I in modo tale che il centro di B corrisponda al pixel in esame. Se il pixel in esame coincide con un contorno allora questo pixel prende valore massimo. Matematicamente la dilatazione è definita dalla formula (1.1.5).

$$I \oplus B = \{p \in Z^2 : p = i + b,\ i \in I \text{ e } b \in B\} \tag{1.1.5}$$

Supponiamo di avere in input la figura 1.1, allora applicando la formula della dilatazione (1.1.5) utilizzando un elemento strutturante lineare, in posizione verticale, si ottiene la figura 1.4. Se la precedente struttura viene sostituita da un elemento strutturante di forma circolare si ottiene l'immagine rappresentata in figura 1.5. Si può notare come le due immagini presentino delle

differenze sostanziali causate proprio dalla forma dell'elemento strutturante utilizzato.
L'erosione, invece, ha il compito di corrodere progressivamente i vari contorni non di sfondo, quindi riduce i bordi degli oggetti e aumenta le dimensioni dei fori. Questo ha il compito opposto della dilatazione. Per effettuare l'erosione abbiamo bisogno, anche questa volta, di un elemento strutturante.
Supponiamo come per la dilatazione di avere l'immagine I e l'elemento strutturante B, allora l'erosione viene effettuata una sovrapposizione di B su I in modo tale che il centro di B corrisponda al pixel in questione. Se tutti i pixel di B appartengono ad un oggetto allora il pixel di I mantiene il suo valore, mentre se un pixel di B appartiene allo sfondo allora il pixel in input prende il valore dello sfondo. Matematicamente l'erosione la si può scrivere come in formula (1.1.6).

$$I \ominus B = \{p \in Z^2 : p + b \in I, \forall b \in B\} \tag{1.1.6}$$

Supponiamo di voler applicare l'erosione alla figura 1.1. Applicando la formula dell'erosione (1.1.6) con elemento strutturante lineare, in posizione verticale, si ottiene l'immagine mostrata in figura 1.6. Se si sostituisce la precedente struttura con un elemento strutturante a forma circolare si ottiene la figura 1.7. Si può notare come anche l'erosione dipenda dalla forma dell'elemento strutturante utilizzato.

Figura 1.4: Esempio di dilatazione con elemento strutturante lineare

Figura 1.5: Esempio di dilatazione con elemento strutturante con forma circolare applicata alla chiesa di San Giorgio

Figura 1.6: Esempio di erosione con elemento strutturante con forma circolare

Figura 1.7: Esempio di erosione con elemento strutturante con forma circolare

1.1.3 Riempimento

Il più semplice algoritmo per il riempimento di una regione è basato su delle combinazioni di funzioni semplici. Supponiamo di lavorare con immagini binarie [4] con valore 0 sullo sfondo, allora si utilizza la formula matematica (1.1.7) per il riempimento dei bordi.

$$X_k = \big(X_{k-1} \oplus B\big) \cap A^c \text{ con } k = 1, 2, 3, ... \tag{1.1.7}$$

dove A è l'immagine di partenza da riempire e B è un l'elemento strutturante simmetrico. Questo algoritmo termina quando $X_k = X_{k-1}$. L'unione di tutte le X_k con A contiene l'immagine con i bordi riempiti. La dilatazione riempirebbe tutta l'area, ma usata con l'intersezione in combinazione con il complementare di A, allora viene riempita solo l'aria di interesse. Questo procedimento prende anche il nome di dilatazione condizionata.
Supponiamo di prendere in considerazione la figura 1.8 sulla quale si vuole applicare il riempimento. Prima di potere utilizzare tale algoritmo si deve avere l'immagine contenente solo i bordi degli oggetti, quindi alla figura in input viene applicato il procedimento di Canny [1.1.1] ottenendo l'immagine 1.9. Avendo solo i bordi è possibile applicare il riempimento così da avere l'immagine rappresentata in figura 1.10.
Non tutte le celle dell'alveare sono riempite, questo fattore è dovuto al fatto che i bordi di alcuni oggetti non sono chiusi.

Figura 1.8: Celle di un alveare

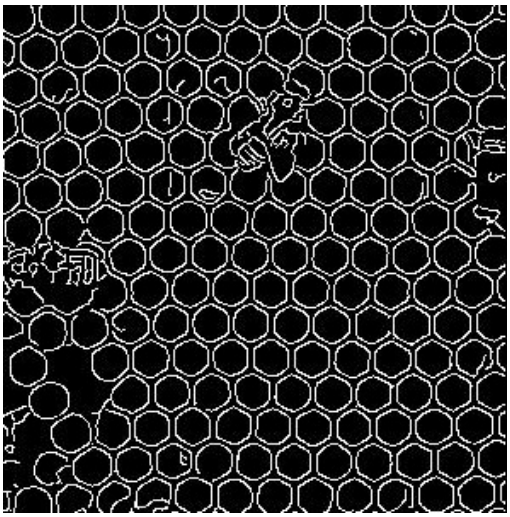

Figura 1.9: Esempio del filtro di Canny applicato ad un alveare: Determina tutti i bordi dei vari oggetti

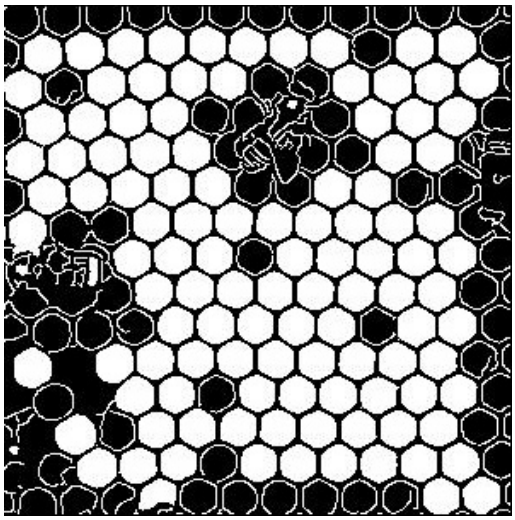

Figura 1.10: Esempio di riempimento bordi applicato alle celle di un alveare: solo i bordi bordi chiusi vengono riempiti

1.1.4 Thresholding

Il thresholding è un metodo per la segmentazione di un'immagine, in cui da un'immagine in scala di grigi in input viene restituita un'immagine binaria in output.
Prima di effettuare tale processo bisogna stabilire un valore di soglia e successivamente confrontare questo valore con tutti i valori dei pixel presenti

nell'immagine. Tutti i pixel con valore maggiore o uguali al valore di soglia vengono sostituiti con un 1, mentre tutti gli altri valori vengono posti a 0. Matematicamente questo procedimento è possibile scriverlo come in formula (1.1.8).

$$I_b = \begin{cases} 0 & I_o < \text{Soglia} \\ 1 & I_o \geq \text{Soglia} \end{cases} \tag{1.1.8}$$

Supponiamo di avere la figura 1.1 da voler binarizzare, allora applicando questo processo, con un valore di soglia 0.6 si ottiene l'immagine mostrata in figura 1.11.

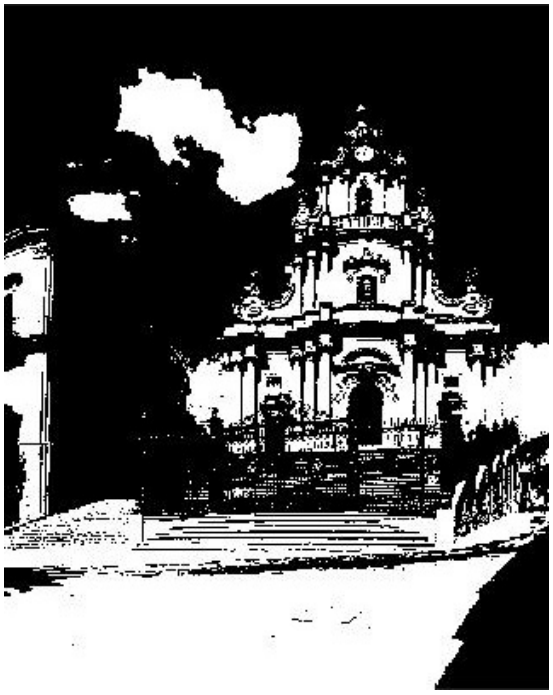

Figura 1.11: Esempio di thresholding: Immagine binarizzata con soglia 0.6

1.2 MATLAB

MATLAB [7], il cui nome deriva da Matrix Laboratory, è un software creato verso la fine degli anni settanta. Ad oggi questo software permette l'elaborazione di matrici, visualizzazione di grafici 2D e 3D, implementazione di algoritmi grazie ai numerosi ToolBox integrati e molto altro. Il suo vasto utilizzo ha portato la sua distribuzione per numerosi sistemi operativi tra

cui MacOs, GNU/Linux e Windows. Per questa tesi viene utilizzato proprio quest'ultima versione.
MATLAB oltre alle numerose funzioni elencate ha un linguaggio di programmazione molto simile ai linguaggi di programmazione più utilizzati. Inoltre, non ha bisogno di dichiarazioni delle variabili, quindi permette di creare matrici con dimensioni non prestabilite e facilmente variabili. Questo rende MATLAB uno dei software più utilizzati negli ambiti ingegneristici e scientifici.
Per lo sviluppo di questa tesi è stato utilizzato MATLAB insieme a l'Image Processing Toolbox. Image Processing Toolbox è un pacchetto software contenente un set di algoritmi per esclusivi per l'elaborazione delle immagini. Con questo pacchetto è possibile effettuare la correzione, riduzione del rumore e molto altro in maniera completamente automatica richiamando alcune funzioni. È possibile velocizzare gli algoritmi eseguendoli su processori multicore e GPU.

1.3 OpenCV

Open Source Computer Vision Library [5], comunemente detto OpenCV, è una libreria open source, ovvero modificabile e di libera distribuzione, sviluppata dalla Intel utilizzata per usi di machine vision. Questa libreria possiede più di 2500 algoritmi al suo interno e possono essere utilizzati per svariati scopi come identificazione di oggetti e volti, tracciamento di movimenti e molto altri.
Questa libreria ha interfacce per vari linguaggi di programmazione, come C++ e Python, ed è facilmente interfacciabile con MATLAB.
In questa tesi è stata utilizzata per sostituire gli algoritmi presenti nell'Image Processing Toolbox durante la trascrizione da MATLAB in C++.

Capitolo 2

Sviluppo dell'algoritmo

In questo capitolo si affronta il ragionamento seguito per lo sviluppo dell'algoritmo, in MATLAB, partendo dalla suddivisione logica del codice [2.1] fino al passaggio dal caso ideale a quello reale [2.4].

2.1 Suddivisione Logica

Il programma logicamente può essere suddiviso in quattro parti:

1. Caricamento dell'immagine e filtraggio della stessa;
2. Determinazione della posizione della piastra e successivo ritaglio della stessa;
3. Ricerca dei vari fori e determinazione della loro posizione nell'immagine;
4. Codifica della matrice;

2.2 Funzioni Rilevanti

Di seguito sono elencate alcune funzioni rilevanti utilizzate per implementare l'algoritmo:

- Imread: Funzione che legge un'immagine dall'indirizzo fornito in ingresso. L'immagine letta viene convertita in una matrice di interi.
- Imerode e imdilate: Queste funzioni hanno il compito, rispettivamente, di erodere e dilatare un'immagine binaria con un elemento strutturante. Sia la figura che l'elemento devono essere passati in ingresso.

- Imrotate: Funzione che ruota in senso antiorario l'immagine di un angolo passato in input. L'applicazione della rotazione avviene nel centro dell'immagine. Se la matrice di partenza non riesce a contenere la nuova matrice ruotata, allora questa viene ridimensionata automaticamente.
- Edge: È una funzione che restituisce un'immagine binaria contente solo i bordi degli oggetti contenuti nell'immagine. Ha bisogno, in ingresso, l'immagine e la tipologia di filtro da utilizzare.
- Imbinarize: Presa in input un'immagine ne restituisce una binaria. Questa funzione sostituisce tutti i valori sopra una certa soglia con un 1 e tutti gli altri con 0. La soglia è determinata automaticamente in maniera globale.
- Imfill: Funzione che riempie i bordi degli oggetti contenuti in un'immagine binaria.
- Imcrop: Questa funzione serve a ritagliare un'immagine. Questo può avvenire in maniera interattiva con l'utente oppure inserendo delle coordinate fissate.
- Bwboundaries: Trova i bordi esterni degli oggetti in un'immagine binaria. Questa funzione riesce anche a distinguere gli oggetti genitori e tracciare i loro figli, ovvero gli oggetti più esterni e gli oggetti completamente racchiusi nei genitori. In uscita restituisce una matrice contenente le posizioni dei bordi, una matrice contenente gli oggetti, il numero di oggetti trovati e una matrice delle adiacenze.
- Regionprops: Questa funzione restituisce delle misurazioni di alcune proprietà specificate in ingresso per ogni componente nell'immagine binaria passata in input. Questi valori vengono ritornati all'interno di una struttura.

2.3 Caso Ideale

Per il caso ideale si suppone che l'input fornito sia un'immagine come quella mostrata in figura 2.1. Infatti, questo tipo di input presenta solo la piastra forata, di colore nero e sfondo bianco, senza alcun tipo di disturbo, come rumore o oggetti estranei. La matrice non deve essere situata necessariamente al centro dell'immagine.

2.3.1 Filtraggio dell'immagine

La prima parte del programma è deputata al caricamento e il filtraggio dell'immagine. L'idea è quella di elaborare l'immagine ideale 2.1, presa in input

fino ad ottenere una nuova immagine, come quella rappresentata in figura 2.3, sulla quale è più semplice effettuare tutte le elaborazioni successive.
Dopo aver caricato l'immagine e convertita in scala di grigi ne viene ottimizzato il contrasto, in maniera automatica, attraverso la funzione imadjust [8]. Ottimizzato il contrasto è possibile iniziare la determinazione degli oggetti attraverso il filtro di Sobel [1.1.1]. Questo processo ritorna una matrice di valori con cifre approssimate al millesimo, quindi è necessario trasformare questa matrice in un'immagine binaria e ciò avviene utilizzando la funzione imbinarize [2.2], ottenendo così l'immagine in figura 2.2. L'ultimo passaggio è quello di riempire i bordi degli oggetti chiusi attraverso la funzione imfill [2.2], così da avere l'immagine 2.3.

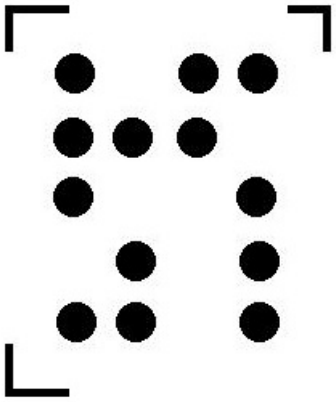

Figura 2.1: Matrice forata nella caso ideale

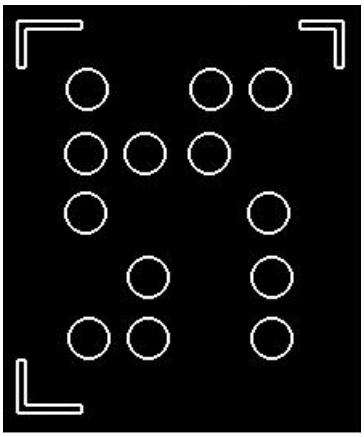

Figura 2.2: Esempio di matrice filtrata: Ottenuta binarizzando il risultato dell'algoritmo di Sobel

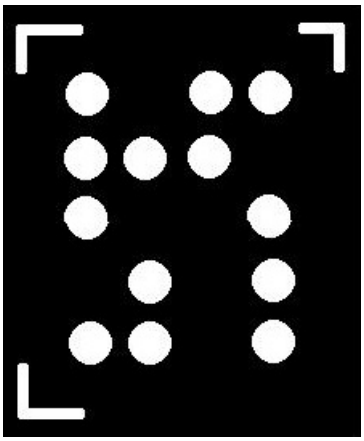

Figura 2.3: Esempio matrice riempita: ottenuta riempiendo gli oggetti chiusi

2.3.2 Ricerca della Matrice

Non è detto che l'immagine riporti la piastra forata al centro, ma la zona di interesse potrebbe essere posizionata in qualsiasi punto dell'immagine, come mostrato in figura 2.4. La seconda parte del programma si occupa quindi di trovare la zona di interesse e successivamente di creare una nuova immagine contenente solo la matrice.
Per effettuare questo procedimento si ha la necessità di prelevare ogni singolo oggetto attraverso la funzione bwboundaries [2.2] e successivamente determinarne la tipologia dell'oggetto prelevato, cioè stabilirne la forma. Per far ciò viene eseguito un algoritmo, il quale come primo passaggio calcola il perimetro, l'area e il centro di ogni oggetto attraverso la funzione regionprops [2.2] e successivamente, sempre per ogni oggetto, calcola una variabile di nome metric con valore dato dalla formula (2.3.1).

$$metric = 4 \cdot \pi \cdot \frac{Area}{Perimetro^2} \tag{2.3.1}$$

Metric prende valore compreso tra 0 e 1, il quale rappresenta la probabilità che l'oggetto sotto esame sia un cerchio. In particolare, gli angoli che delimitano la zona di interesse hanno un valore compreso tra 0.1 e 0.35. Un esempio di questi valori è rappresentato in figura 2.5.
Grazie a questo procedimento è possibile determinare e salvare la posizione degli angoli che racchiudono la matrice così da poter ritagliare l'immagine, dipendentemente dalle loro coordinate, e crearne una nuova contenente solo la matrice da elaborare. Il ritaglio dell'immagine viene effettuato utilizzando la funzione imcrop [2.2] passando in ingresso le coordinate degli angoli, ottenendo l'immagine in figura 2.6.

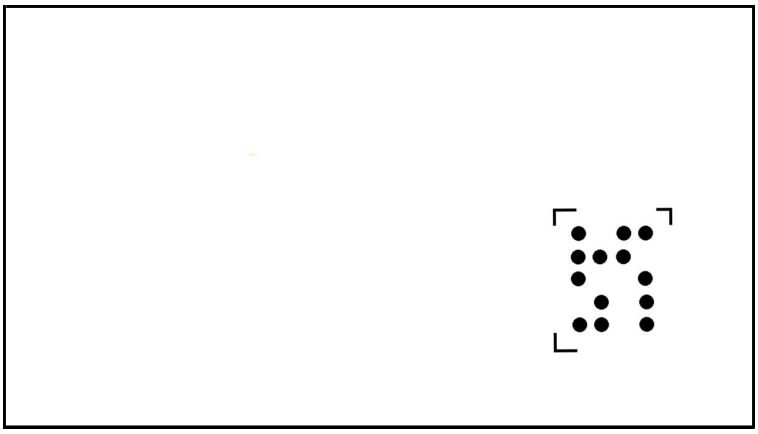

Figura 2.4: Matrice forata nella caso ideale situata in un qualsiasi punto

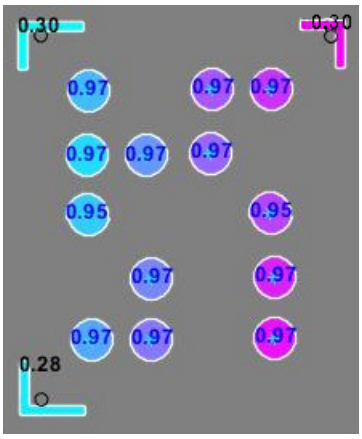

Figura 2.5: Esempio di matrice filtrata: I valori rappresentano il parametro metric

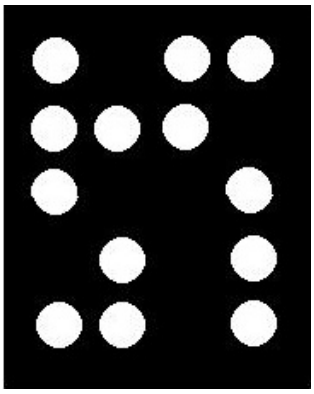

Figura 2.6: Esempio matrice ritagliata: ottenuta tagliando dalle coordinate degli angoli

2.3.3 Ricerca dei fori

Avendo l'immagine 2.6, la quale contiene solo la zona di interesse, è necessario determinare tutti i fori e la loro posizione spaziale. La terza parte del programma utilizza un algoritmo simile a quello visto al precedentemente [2.3.2], infatti vengono prelevati tutti gli oggetti con la funzione bwboundaries e per ognuno di essi viene calcolato il proprio centro e il valore metric, come mostrato in figura 2.7.
Successivamente vengono salvati tutti i centri degli oggetti che presentano un valore di metric compreso tra 0.7 e 1. Viene scelto come valore minimo 0.7 poiché i filtri potrebbero distorcere i fori rendendoli forme simili a cerchi, un esempio di ciò lo si può vedere nella figura 2.8, dove i fori di colore blu rappresentano proprio i cerchi deformati.

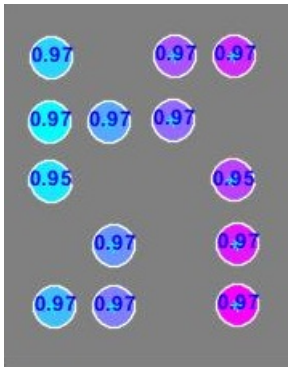

Figura 2.7: Esempio di matrice che presenta i valori metric

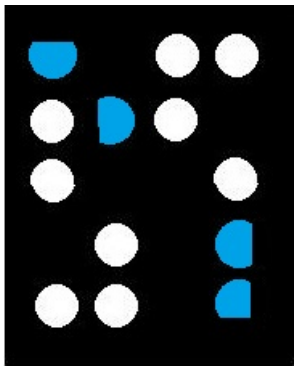

Figura 2.8: Esempio matrice deformata: i fori di colore blu sono quelli che presentano una deformazione dovuto al filtraggio

2.3.4 Codifica della Matrice

L'ultimo passaggio dell'algoritmo è quello di codificare la matrice. Per prima cosa viene effettuato il conteggio del numero di righe e colonne della matrice forata. Subito dopo viene creata una nuova matrice di soli zeri con dimensioni il numero di righe e colonne appena trovati.
Determinate le righe e le colonne è possibile tracciare una griglia dove gli incroci delle righe con le colonne rappresentano il punto in cui è possibile trovare un foro. Un esempio di questa griglia è rappresentato in figura 2.9. È possibile calcolare la distanza, in pixel, di ogni colonna rispetto all'angolo in alto a sinistra, detto angolo d'origine. Stesso ragionamento per le righe. Tutto ciò serve ad associare il numero di riga, o colonna, con la sua distanza dall'angolo d'origine.
Questo procedimento serve ad associare ogni foro con il proprio numero di riga e colonna, così da trovare la sua posizione all'interno della griglia, mostrata in figura 2.9. La griglia però corrisponde alla matrice di codifica, quindi trovando la posizione del foro sulla griglia si trova l'ubicazione del foro all'interno della matrice.
Tutte le celle della matrice di codifica contenenti un foro hanno valore 1, mentre tutte le altre prendono valore 0.

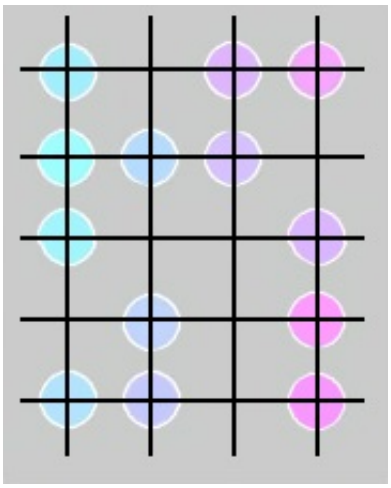

Figura 2.9: Griglia che rappresenta la posizione dei fori

2.4 Dal caso ideale a quello reale

Per effettuare il passaggio dal caso ideale al caso reale vi sono delle importanti modifiche da effettuare. La prima considerazione sta nelle caratteristiche dell'immagine stessa. Essa potrebbe infatti presentare alcuni problemi tra cui la messa a fuoco della videocamera, oggetti estranei alla matrice, essere rotata, oppure l'immagine potrebbe avere del rumore. Tenendo in considerazione tutti questi problemi sarà quindi necessario una apposita elaborazione dell'immagine. Inoltre, ulteriori problemi possono sorgere nella decodifica a causa di errori, ombre e tolleranze.

2.4.1 Filtraggio Reale

Il filtraggio avviene similmente a quanto visto nel capitolo [2.3.1], ma vi sono alcune differenze sostanziali. Infatti, dopo avere caricato l'immagine questa viene immediatamente filtrata attraverso un filtro Gaussiano per rimuovere il rumore. Per la determinazione dei bordi degli oggetti è necessario usare un filtro di Canny rispetto ad un Sobel, perché il metodo di Canny fornisce una determinazione dei bordi più accurata. Il codice prosegue normalmente fino a dopo aver riempito gli oggetti. Successivamente si ha la necessità di erodere l'immagine per poi dilatarla portandola nuovamente allo stato prima dell'erosione. Viene effettuato questo procedimento poiché l'erosione permette di eliminare tutti gli elementi piccoli lasciando solo oggetti più grandi e successivamente la dilatazione permette di far tornare gli elementi allo stato originario, ma comunque lasciando eliminati gli elementi più piccoli. Per effettuare queste due operazioni si ha la necessità di avere un elemento strutturante, che in questo caso prende una forma di circolare con

raggio 3. Infine, l'immagine viene ruotata se necessario.
Supponiamo di avere un'immagine da ruotare, come quella mostrata in figura 2.10, allora vengono cercati gli angoli che delimitano la matrice così da poter effettuare la rotazione. Una volta scovati, essi vengono salvati in un nuovo array così da poter ricercare la posizioni corrispondenti ai valori minimo e massimo della x. Lo stesso ragionamento si applica per la y. Una volta determinati questi valori viene applicata la formula (2.3.2) per determinare l'angolo, in radianti, della rotazione da effettuare.

$$\theta = \arctan\left(\frac{Y_{max} - Y_{min}}{X_{max} - X_{min}}\right) \tag{2.3.2}$$

Per applicare questo processo viene utilizzata la funzione imrotate [2.2] la quale necessita in input, oltre l'immagine da ruotare, l'angolo espresso in gradi, quindi è necessario effettuare una conversione dell'angolo da radianti a gradi.
L'angolo potrebbe essere molto piccolo, quindi tutti gli angoli con valori minori o uguali a 5° vengono scartati perché un angolo con queste caratteristiche non influenza la corretta codifica e quindi non è necessario effettuare la rotazione.
Seguendo questo algoritmo il risultato dell'immagine in figura 2.10 è illustrato nell'immagine 2.11.

Figura 2.10: Esempio di matrice da ruotare

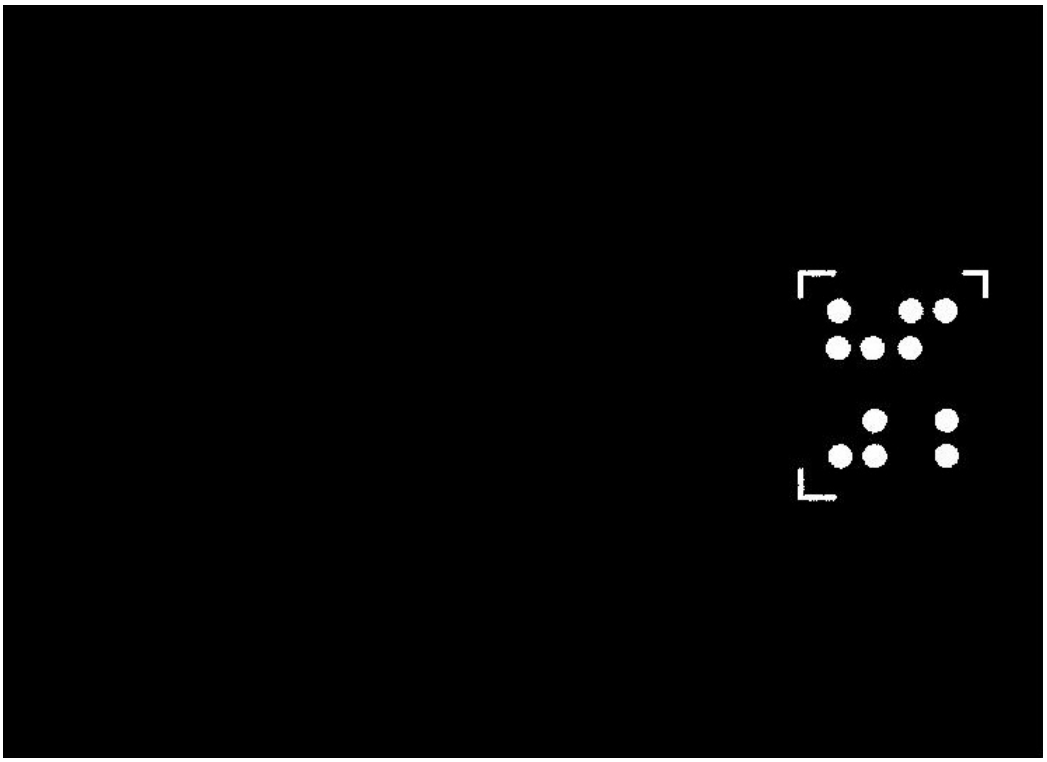

Figura 2.11: Esempio di matrice ruotata

2.4.2 Codifica reale

Il problema principale è che i fori potrebbero non avere i centri allineati, come mostrato in figura 2.12, e potrebbero anche essercene alcuni non disposti in maniera corretta. Il caso peggiore è che tutti i cerchi non siano allineati tra loro, sia sulle righe che sulle colonne.
In questa situazione il calcolo delle righe e delle colonne porterebbe spesso ad un risultato errato. Per risolvere questo problema viene inserita una tolleranza sui valori delle posizioni dei cerchi. Utilizzando questo metodo non viene controllata effettivamente la posizione del cerchio, bensì che il foro sia compreso tra due posizioni. Questo permette di contare un corretto numero di righe e di colonne. Lo stesso procedimento viene utilizzando nel confronto per determinare la posizione dei fori, così da riuscire a codificare correttamente la matrice.

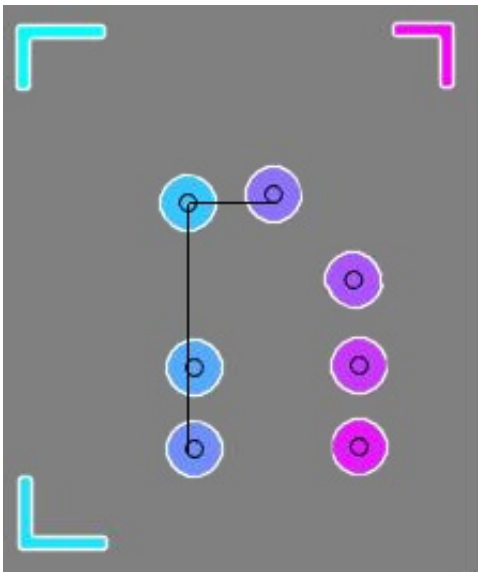

Figura 2.12: Esempio di matrice con i fori non allineati: Preso come riferimento il primo foro si notano che i fori più adiacenti non sono perfettamente allineati

Capitolo 3

Da MATLAB a C++

Per la trascrizione del programma da MATLAB a C++ viene utilizzata la libreria OpenCV [5] tramite la quale è possibile implementare le operazioni di Image Processing. Questa libreria non contiene tutte le funzioni presenti in MATLAB, perché MATLAB presenta alcune funzioni frutto di combinazioni di più algoritmi. OpenCV, invece, al suo interno contiene solamente algoritmi base e nessuna delle loro combinazioni, di conseguenza alcune funzioni sono state implementate al bisogno.

3.1 Funzioni e Variabili Rilevanti

Di seguito sono elencate alcune funzioni e variabili rilevanti utilizzate per implementare l'algoritmo in C++:

- Vector: Questa è una variabile simile ad un array. La differenza sostanziale tra un vettore e un array è che nei vettori è possibile variare dinamicamente le sue dimensioni. Spesso questa variabile viene utilizzata per l'archiviazione di elementi. Come per gli array, i vettori necessitano la dichiarazione del tipo ed è possibile creare vettori multidimensionali.
- Mat: È una variabile che forma una matrice. Questa contiene i valori dei pixel di un'immagine, la quale può essere passata in input oppure creata manualmente. In particolare, questa variabile è formata da due parti, quali intestazione contenente le informazioni relative all'immagine e il puntatore alle sue celle di memoria.
- GaussianBlur: Filtro di Gauss [2] per la rimozione del rumore. Questa funzione in input necessita dell'immagine da filtrare e le dimensioni del filtro da utilizzare.

- Canny: Funzione contenuta nella libreria che, attraverso il metodo di Canny [1.1.1], ha il compito di trovare tutti i contorni degli oggetti contenuti nell'immagine. Questa funzione ha bisogno di alcuni parametri in input, tra cui l'immagine sulla quale determinare i bordi, i valori di soglia del threshold [1.1.8] e il valore del filtro.
- getStructuringElement: Funzione che restituisce un elemento strutturante utilizzato per operazioni morfologiche di immagini 2D. La sua forma e la sua dimensione vengono passate in input.
- Erode e Dilate: Funzioni usate, rispettivamente, per erodere e dilatare l'immagine. Per usare queste funzioni serve un elemento strutturante.
- warpAffine: È una funzione che applica una trasformazione sull'immagine. In particolare, è stata utilizzata per effettuare una rotazione dell'immagine. Questa trasformazione avviene come prodotto tra matrici.
- floodFill: Funzione che riempie un oggetto chiuso. L'oggetto viene riempito con un colore fornito in ingresso. Il punto di partenza dell'oggetto da cui far partire il riempimento deve essere fornito in ingresso.
- approxPolyDP: Funzione che approssima le curve degli oggetti in un'immagine con una precisione che viene passata in input. L'immagine con gli oggetti approssimati viene fornita in output.

3.2 Strutture

In C++ o simili linguaggi, una struttura dati è un modo per raggruppare logicamente un insieme di variabili anche con tipi diversi, in una unica dichiarazione. Per dichiarare una nuova struttura si segue la seguente definizione:

```
1   struct nome_struttura{
2           tipo1 nome_variabile1;
3           tipo2 nome_variabile2;
4           .
5           .
6           .
7   };
```

dove il nome_struttura rappresenta proprio il nome della struttura, il quale può essere utilizzato come un nuovo tipo di dati. Ad esempio:

```
struct modelli_auto{
        char nome[30];
        int anno;
        float chilometri;
        char descrizione[500];
};

modelli_auto Audi_TT;
modelli_auto Citroen_DS3, Subaru_Impreza;
```

Tra le parentesi graffe sono indicati tutti i tipi di variabili che costituiscono la struttura. Questi dati possono anche essere richiamati singolarmente, senza il bisogno di richiamare l'intera struttura.
Per questo algoritmo sono state utilizzate le seguenti strutture.

3.2.1 Struttura RC

Questa struttura è formata da soli due elementi con lo stesso tipo. Questi sono due vettori di tipo "int" utilizzati per memorizzare le righe e le colonne della matrice da codificare.
È stato scelto di creare questa struttura perché si è ritenuto opportuno tenere separate le righe con le colonne in due vettori differenti così che possano avere anche dimensioni diverse, ma tenerle comunque raggruppate sotto un unica variabile.

```
struct rc {
        vector<int> rg;
        vector<int> cl;
};
```

3.2.2 Struttura Immagine

La struttura immagine viene utilizzata per immagazzinare informazioni riguardante l'immagine da elaborare. In particolare, la struttura salva sia la figura che gli oggetti che la compongono. Inoltre, insieme agli oggetti vengono salvati anche le loro proprietà come la posizione dei centri di ogni oggetto e il loro valore metric.

```
struct immagine {
        vector<vector<Point>> contours;
        vector<vector<Point>> contours_poly;
        vector<Point2f> centers;
        Mat contourOutput;
        double metric[100];
};
```

3.2.3 Struttura Elementi

L'ultima struttura prende il nome di elementi. In questa struttura vengono memorizzati solo le posizioni degli oggetti che rappresentano i fori o gli angoli che delimitano la zona d'interesse. Inoltre, per gli angoli vengono anche immagazzinati i valori minimi e massini sia delle X che per le Y.

```
struct elementi {
        vector<vector<Point>> Angoli;
        vector<Point2f> Centri;
        int Valori[4];
        int Valori2[4];
};
```

3.3 Funzioni Implementate

Non tutte le funzioni di MATLAB sono presenti nella libreria OpenCV. È stato quindi necessario implementarle opportunamente in maniera tale da riprodurne il funzionamento.

3.3.1 imRotate

Questa funzione è l'algoritmo equivalente del metodo imrotate [2.2] presente su MATLAB. In generale, la funzione ha il compito ruotare l'immagine passata in input di un angolo, anche esso passato in ingresso.
All'inizio l'algoritmo presenta un controllo, il quale verifica se l'angolo in input è un multiplo di 360° così da comprendere se l'immagine effettivamente necessita di una rotazione.
Se dal controllo l'angolo non risulta un multiplo dell'angolo giro, allora viene effettuata una rotazione con applicazione il centro dell'immagine. Effettuata la rotazione viene creato un rettangolo, il quale viene sovrapposto alla matrice ruotata e ne viene controllata se vi è una perdita di informazioni. Se questo dovesse accadere, allora viene creata una nuova matrice, con dimensioni maggiori alla precedente, che riesca a contenere la nuova immagine.
Quando l'algoritmo è terminato, questo ritorna una matrice di tipo Mat contenente la nuova immagine ruotata.

L'algoritmo è mostrato nell'appendice A.1.

3.3.2 Imfill

Come suggerisce anche il nome, questa è la funzione analoga a Imfill [2.2] di MATLAB.
Infatti, Imfill è una funzione utilizzata per riempire gli oggetti chiusi, la quale necessita in input un'immagine. Prelevata la figura da elaborare viene avviato un ciclo per ogni colonna dell'immagine e al suo interno vi sono due controlli separati. Il primo controllo verifica che il valore situato nella posizione $(0, i)$ abbia valore uguale a 0 e se questo è verificato, allora viene utilizzata la funzione floodFill [3.1] nel punto $(i, 0)$ per riempire l'oggetto. Il secondo controllo effettua lo stesso controllo, ma nella posizione $(ultimariga, i)$ e il punto di applicazione del floodFill è $(i, ultimariga)$.
Concluso il ciclo viene effettuato il negativo dell'immagine riempita per poi essere ritornata in uscita. L'algoritmo è visibile sull'appendice A.2.

3.3.3 Trova Valori

Questa è una funzione di tipo elementi che svolge un procedimento simile a quello visto al capitolo [2.3.3]. In generale, questa funzione viene utilizzata per trovare le posizioni del centro di ogni foro e la posizione degli angoli che delimitano la zona d'interesse. Ogni valore trovato viene salvato in una variabile di tipo elementi [3.2.3]. In particolare, questa funzione, oltre a salvare la posizione dei centri di ogni cerchio, viene utilizzata per trovare e salvare la posizione degli angoli che delimitano la matrice. È di particolare interesse trovare la loro posizione perchè viene utilizzata per effettuare successivamente il ritaglio, sempre se ciò risulta necessario. Questa ricerca viene effettuata adoperando il parametro metric, visto nel capitolo [2.3.3].
Tra le posizioni degli angoli precedentemente memorizzate, vengono cercati i pixel con coordinate minime e massime, sia per l'asse delle X che per l'asse delle Y. Per determinare questi valori viene usato un algoritmo di comparazione. Alla fine di questo ciclo si avranno i pixel desiderati che verranno utilizzati per effettuare un ritaglio dell'immagine contenente la zona di interesse nella maniera più accurata possibile.
Alla fine di questo processo la funzione ritorna una variabile di tipo elementi. L'algoritmo è mostrato nell'appendice A.3.

3.3.4 Trova bordi

Trova bordi è una funzione, di tipo immagine [3.2.2], utilizzata per prelevare tutti gli oggetti dall'immagine e calcolare le proprietà degli oggetti prelevati. In generale, dall'immagine passata in input la funzione preleva ogni oggetto e ne salva i bordi in un vettore. Successivamente per ogni oggetto calcola il

suo centro e il suo coefficiente metric utilizzando la formula di tipo immagine (2.3.1).
Terminati tutti i processi, allora i valori determinati vengono salvati in una variabile di tipo immagine con nome ritorno e successivamente viene fornita in uscita.
L'algoritmo è mostrato nell'appendice A.4.

3.3.5 Righe Colonne

La funzione righe_colonne è una funzione di tipo RC [3.2.1] la quale ha il compito di determinare il numero di righe e colonne e le loro coordinate rispetto all'angolo d'origine.
Questa funzione svolge un compito simile alla codifica reale [2.4.2]. Infatti, determina la coordinate delle righe e colonne, ma non effettua la codifica. Anche in questo caso viene utilizzata una tolleranza dato che i fori potrebbero non essere allineati.
L'algoritmo è visibile nell'appendice A.5.

3.4 Descrizione del programma

Il programma ricalca dal punto di vista logico la versione MATLAB, ovviamente con le dovute differenze legate all'uso del linguaggio C++. Questa trascrizione è stata effettuata in ambiente Windows utilizzando il software "Visual Studio" nel quale poi è stata installata la libreria OpenCV.
Caricata in buffer l'immagine il processo inizia con il filtraggio, che avviene similmente a quanto visto nel capitolo [2.4.1].
Filtrata l'immagine, questa necessita di essere binarizzata. Fatto ciò inizia il processo di ricerca della matrice forata all'interno dell'immagine. Il primo passaggio effettuato è la ricerca degli angoli, attraverso la funzione [3.3.3], per calcolare l'angolo di rotazione, se necessario, e successivamente ritagliare la zona d'interesse.
Ottenuta solo la matrice si inizia la ricerca dei fori. Si procede cercando tutti gli oggetti presenti nell'immagine ritagliata attraverso la funzione con nome trova bordi [3.3.4].
Salvate le posizioni dei fori è possibile stabilire le dimensioni della zona d'interesse utilizzando la funzione [3.3.5].
L'ultimo passaggio rimasto è la codifica della matrice. Questa codifica avviene in maniera analoga a quanto visto nel capitolo [2.4.2].
All'inizio del programma viene utilizzata la funzione clock, la quale viene nuovamente richiamata alla fine del programma. Questa è una funzione della libreria "time.h" [3], che ritorna il numero di cicli di clock dall'ultima chiamata della funzione. Se all'inizio del codice si salva, in una variabile [T_0],

il valore restituito dalla funzione clock() e in un'altra variabile $[T_1]$ si memorizza il valore restituito nuovamente della funzione clock(), la quale viene richiamata alla fine del codice, allora si può ottenere il tempo di esecuzione utilizzando la formula (3.4.1).

$$tempo = \frac{T_1 - T_0}{CLOCK\_PER\_SEC} \tag{3.4.1}$$

Dove "CLOCK_PER_SEC" fornisce il numero di clock per secondo a seconda della macchina in cui il codice è stato eseguito.
Un esempio dell'uso di clock è mostrato alla fine della sezione.

```
1   int main() {
2       clock_t start, end; //definisco il tempo
3       double tempo;
4       start = clock(); //inizio a contare
5
6       OPERAZIONI
7
8       end = clock();
9       tempo = ((double)(end - start)) / CLOCKS_PER_SEC;
10      printf("\nIl tempo di esecuzione totale e' di circa: %f Secondi\n", tempo);
11      return 0;
12  }
```

3.5 Differenze tra MATLAB e C++

In questa sezione si mettono in luce le differenze che vi sono tra il filtraggio effettuato attraverso MATLAB e quello effettuato utilizzando la libreria OpenCV.
Supponiamo di prendere in input la figura 3.1 su cui effettuare il filtraggio. In particolare, questa operazione viene effettuata prima con MATLAB ottenendo la figura 3.2 e successivamente le stesse operazioni vengono effettuate in C++, con l'utilizzo della libreria, ottenendo la figura 3.3.
Queste due immagini presentano delle differenze sostanziali. Infatti MATLAB elimina quasi tutte le ombre, presenti nell'immagine originale, lasciando un solo oggetto superfluo, di dimensioni ridotte, situato in basso a destra. Con il filtraggio attraverso OpenCV, invece, oltre all'oggetto superfluo, rimangono tutte le ombre che circondano la piastra.
Si può notare anche che la zona d'interesse su MATLAB, in particolare gli angoli che delimitano la matrice, vengono completamente rovinati rendendo la ricerca della piastra più difficoltosa. Invece, su OpenCV, la matrice ottenuta dopo il processo è in perfetto stato.

Queste differenze di filtraggio sono dovute alle differenti implementazioni degli algoritmi tra MATLAB e OpenCV.

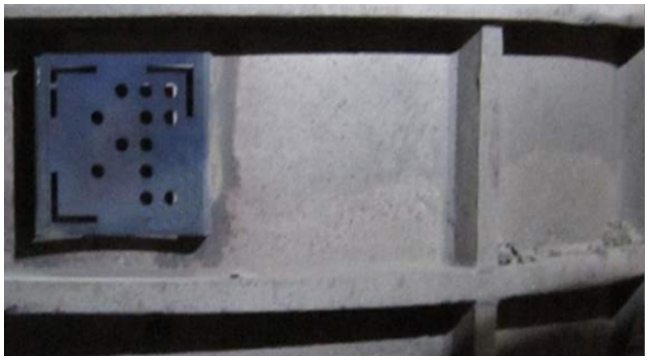

Figura 3.1: Esempio di matrice reale: utilizzata per confrontare le differenze di filtraggio tra MATLAB e OpenCV

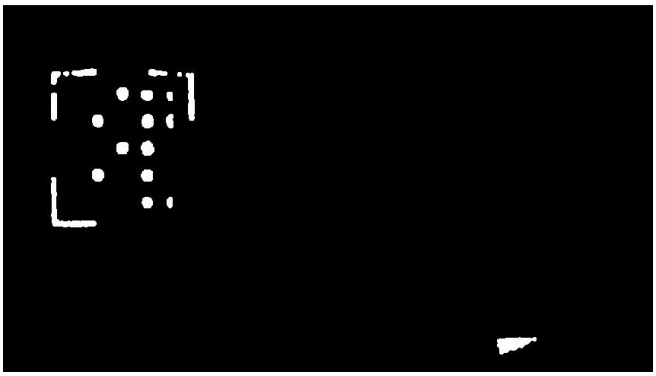

Figura 3.2: Esempio di matrice elaborata con MATLAB

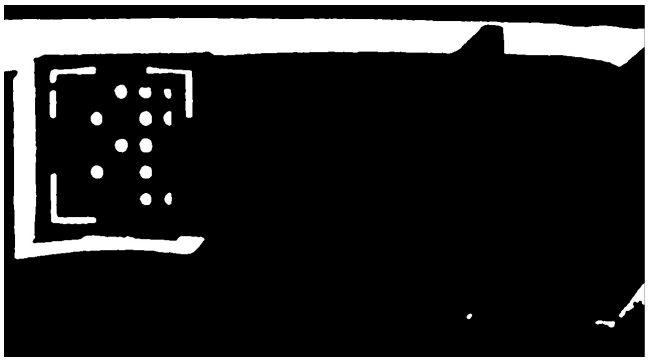

Figura 3.3: Esempio di matrice elaborata in C++

Capitolo 4

Fase di Test

In questo capitolo vengono presentati i vari test effettuati per il collaudo dell'algoritmo. Ogni test effettuato ha come obiettivo la corretta codifica dell'immagine nel minore tempo possibile. Questo non risulta semplice dato che la qualità dell'input potrebbe variare.
I test effettuati posso essere suddivisi per i seguenti scopi:

- Corretta codifica avendo input differenti;
- Test di velocità del codice;

Per ogni simulazione il programma restituisce varie immagini ottenute dopo l'applicazione di ogni filtro, dei valori numerici di debug, e per finire la matrice della codifica. Per migliorare la chiarezza del testo saranno riportate solo le immagini e i valori più significativi.
Ogni test presenta una descrizione iniziale, le varie figure ottenute, eventuali considerazioni importanti.

4.1 Test con immagini differenti

4.1.1 Primo caso: Semplice

Preso in input la figura 4.1 si ci ritrova in uno dei casi migliori su cui lavorare. Questo è dovuto dal fatto che la matrice è posizionata al centro dell'immagine ed inoltre l'immagine è perfettamente dritta. Un'altra caratteristica importante è l'assenza di oggetti estranei. Si possono notare alcune ombre, ma queste non disturbano l'elaborazione dell'immagine.
Il filtraggio avviene come visto nel capitolo [2.4.1]. Durante questo processo vengono generate alcune immagini; in particolare, applicando il filtro di Canny [1.1.1] all'immagine in input si ottiene la figura 4.2. Successivamente una volta trovati tutti gli oggetti questi vengono riempiti utilizzando la

funzione imfill [3.3.2] ottenendo l'immagine mostrata in figura 4.3. Infine, a questa immagine vengono applicati i filtri erosione e dilatazione ottenendo l'immagine in figura 4.4. Per un filtraggio migliore si utilizza un elemento strutturante a forma ellittica invece di uno a forma circolare, così che il risultato si avvicina di più al caso ideale. Per notare queste differenze basta confrontare la figura 4.4 con la figura 4.5.
Tra la figura 4.3 e la figura 4.4 si possono notare delle differenze. Infatti, la prima contiene degli elementi superflui, ma grazie all'applicazione degli ulteriori filtri, questi vengono eliminati lasciando solamente la box con i fori. Questo passaggio giustifica tutti i filtri utilizzati.
Finito il filtraggio inizia la seconda parte del codice, vista al capitolo [2.3.3], con il quale vengono ricercati gli oggetti presenti nell'immagine attraverso la funzione trova bordi [3.3.4]. Determinati tutti gli oggetti, allora essi vengono prelevati eliminando tutto il background. Il risultato di queste operazioni è mostrato in figura 4.6.
Salvati tutti gli oggetti inizia il lavoro di ritaglio e successivamente di codifica. Questo avviene distinguendo la forma degli oggetti utilizzando la formula (2.3.1), la quale rappresenta il parametro metric. Trovato gli oggetti che delimitano la zona d'interesse si effettua il ritaglio.
Dall'immagine ritagliata, visualizzata in figura 4.7, vengono determinate le varie posizioni dei fori segnandone il centro; questo passaggio è messo in evidenza nella figura 4.8. In particolare, grazie a questo procedimento si riesce ad effettuare la codifica ottenendo una matrice formata di 0 e 1 stampata su schermo. La codifica dell'immagine 4.1 è visibile in figura 4.9.

Figura 4.1: Immagine reale presa in input per il primo test

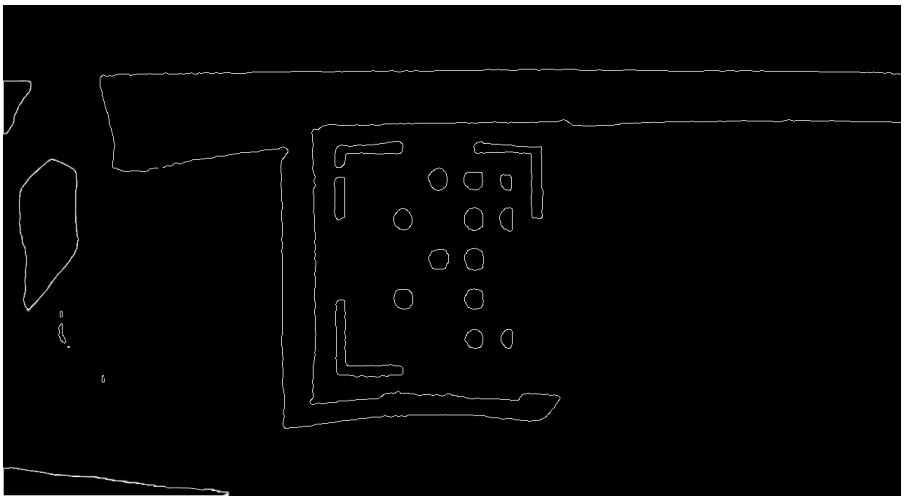

Figura 4.2: Immagine reale filtrata con Canny per il primo test

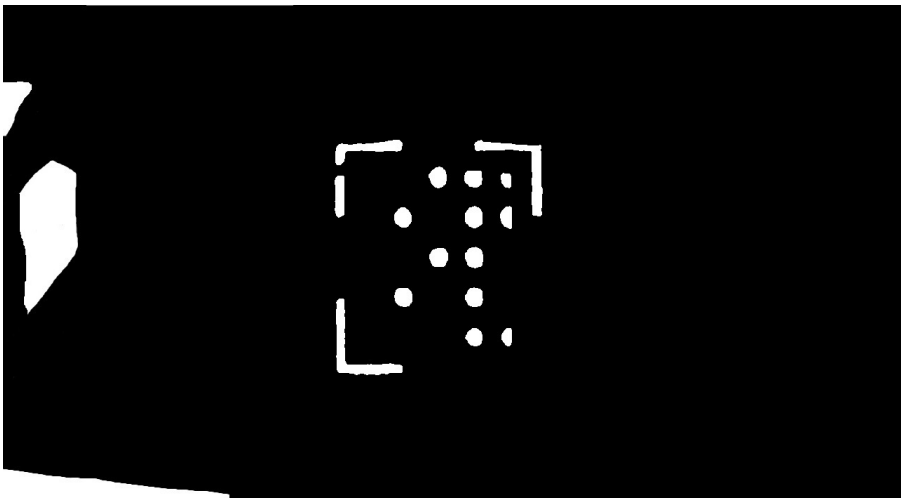

Figura 4.3: Immagine reale filtrata e riempita per il primo test

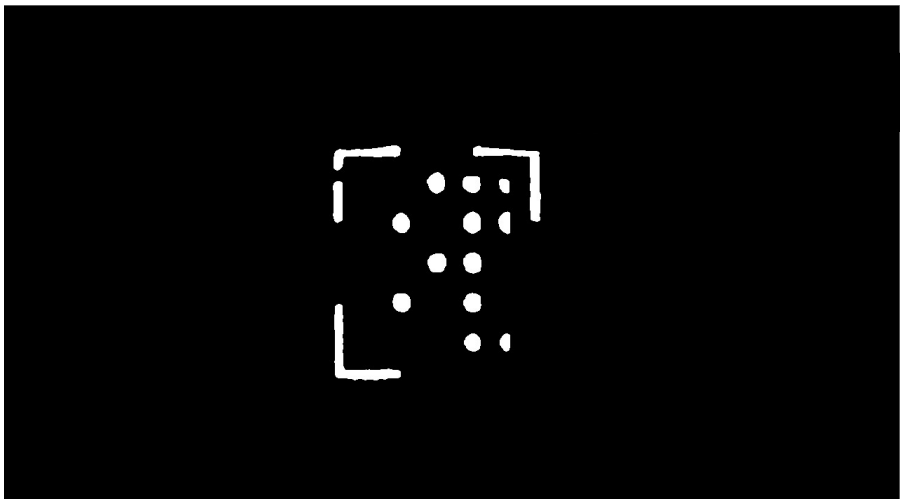

Figura 4.4: Immagine reale erosa e dilatata: viene erosa e dilatata utilizzando un elemento strutturante con forma ellittica

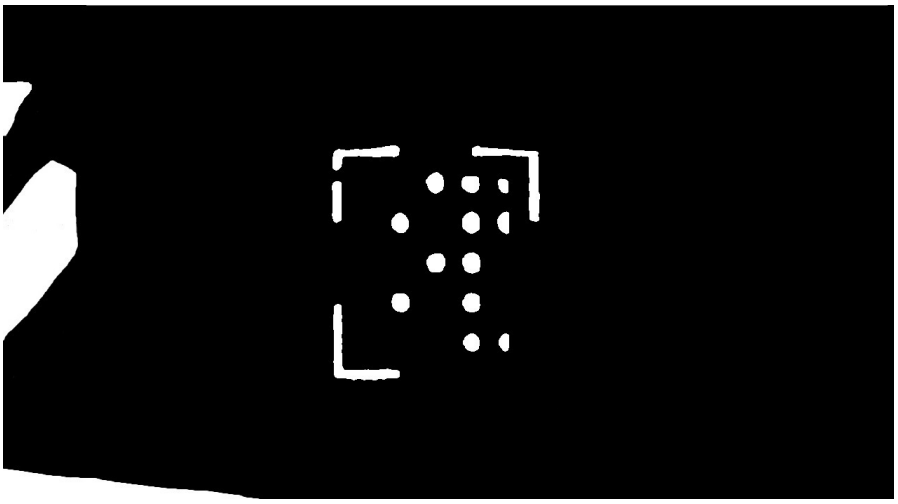

Figura 4.5: Immagine reale erosa e dilatata: viene erosa e dilatata utilizzando un elemento strutturante con forma circolare

Figura 4.6: Immagine reale con oggetti in evidenza: vengono evidenziati tutti gli oggetti eliminando completamente lo sfondo

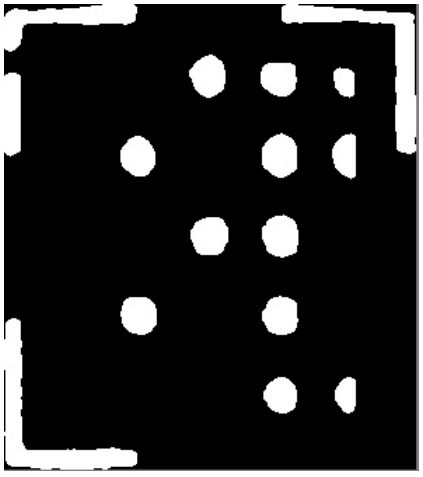

Figura 4.7: Immagine reale ritagliata: taglio effettuato nelle coordinate degli angoli che delimitano la matrice

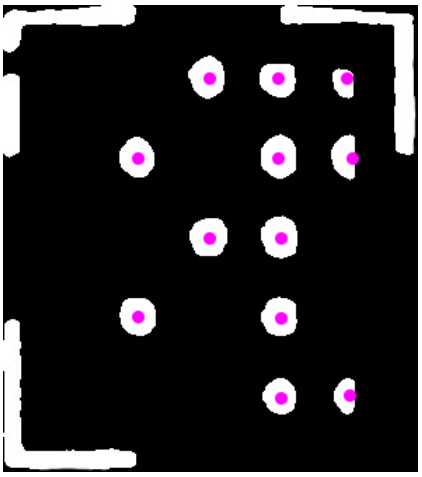

Figura 4.8: Immagine reale mettendo in evidenza i centri dei fori

```
STAMPA MATRICE CODIFICATE
        0       0       1       1       1
        0       1       0       1       1
        0       0       1       1       0
        0       1       0       1       0
        0       0       0       1       1
```

Figura 4.9: Codifica dell'immagine reale del primo caso

4.1.2 Secondo caso: Medio

Un caso differente può essere quello mostrato in figura 4.10 dato che la matrice non è posizionata al centro, ma è situata nell'angolo in alto a sinistra. Durante il filtraggio viene evidenziata una differenza sostanziale rispetto 4.2. Infatti, come si vede in figura 4.11 vi sono diversi elementi superflui che per il codice sono considerati come oggetti e quindi potrebbero influire nel processo di codifica.
In realtà questi oggetti di disturbo non andranno ad influire sulla codifica. Tale affermazione è vera perché anche con la loro presenza si passa comunque alla ricerca degli angoli che delimitano la matrice per poi effettuare un taglio dell'immagine e dato che ognuno di questi elementi di disturbo ha valori di metric (2.3.1) che non rientrano tra le tolleranze fissate, allora l'immagine risultante sarà solamente la zona d'interesse con i fori, come in figura 4.12.
Da questo punto in poi la codifica avviene senza alcun intoppo ottenendo la codifica in figura 4.13 che rappresenta esattamente l'input.
Questo test mostra il corretto funzionamento dell'algoritmo di ricerca della piastra all'interno dell'immagine. Infatti, si nota che l'immagine non deve

avere necessariamente la matrice in posizione centrale, ma che può essere posizionata in qualsiasi punto dello spazio.

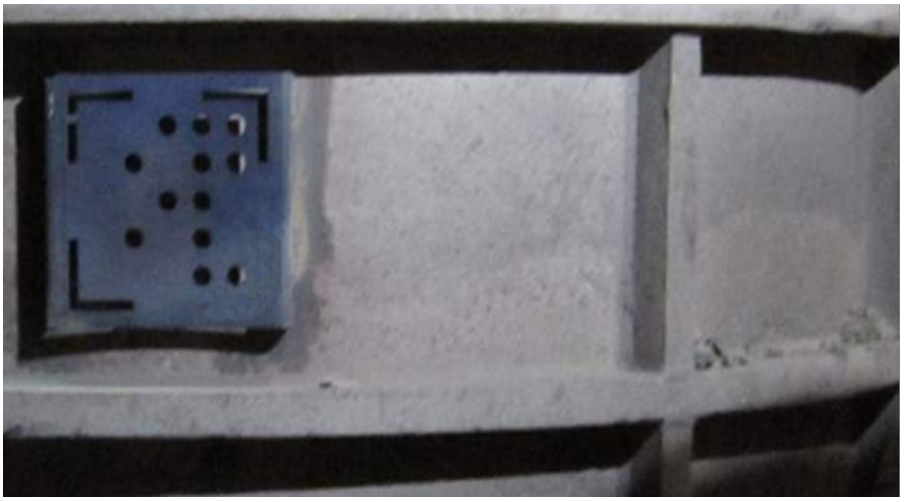

Figura 4.10: Esempio di immagine del secondo caso

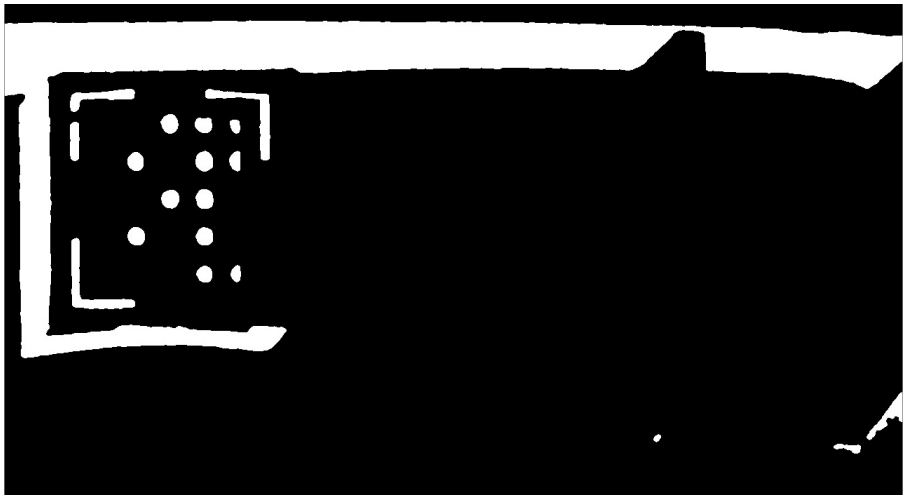

Figura 4.11: Esempio di filtraggio completo del secondo caso

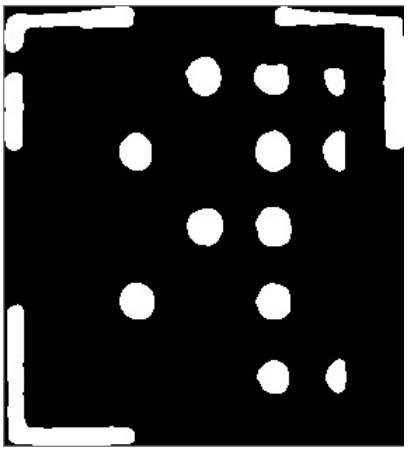

Figura 4.12: Immagine del secondo caso ritagliata

```
STAMPA MATRICE CODIFICATE
        0       0       1       1       1
        0       1       0       1       1
        0       0       1       1       0
        0       1       0       1       0
        0       0       0       1       1
```

Figura 4.13: Codifica dell'immagine reale del secondo caso

4.1.3 Terzo caso: Difficile

L'ultimo caso è quello mostrato partendo dall'input in figura 4.14. Questo è uno dei casi peggiori su cui lavorare perché non solo l'immagine non presenta la matrice centrale, ma questa è ruotata e presenta numerose ombre.
Il primo passaggio da effettuare rimane comunque il filtraggio dato che ci permette di eliminare tutti gli elementi superflui. Come si vede in figura 4.15 il filtraggio rimuove quasi tutto lasciando solo la zona di interesse ed un solo elemento estraneo.
Ottenuta questa immagine è possibile raddrizzarla attraverso l'algoritmo illustrato nel capitolo 2.4.1, il quale determina la posizione degli angoli e attraverso la formula (2.3.2) determina l'angolo di rotazione da effettuare così da ottenere l'immagine 4.16.
Ruotata la figura si passa alla ricerca dei vari angoli per effettuare un taglio, così da rimuovere l'oggetto superfluo che comunque è rimasto nell'immagine anche se ruotato. Successivamente l'oggetto superfluo non viene più tenuto in questione perché il suo valore metric (2.3.1) ha un valore esterno ai valori fissati.

Con il taglio si ottiene la sola zona di interesse, come in immagine 4.17, la quale contiene solamente la matrice senza alcun elemento estraneo. Da questo punto in poi si effettua il processo di codifica dell'immagine ritagliata ottenendo così la matrice di codifica mostrata in figura 4.18.
Come esito si ha il fatto che anche se la piastra non è dritta, ma presenta una rotazione come in questo caso è comunque possibile effettuare una corretta codifica grazie all'algoritmo sviluppato per la rotazione.

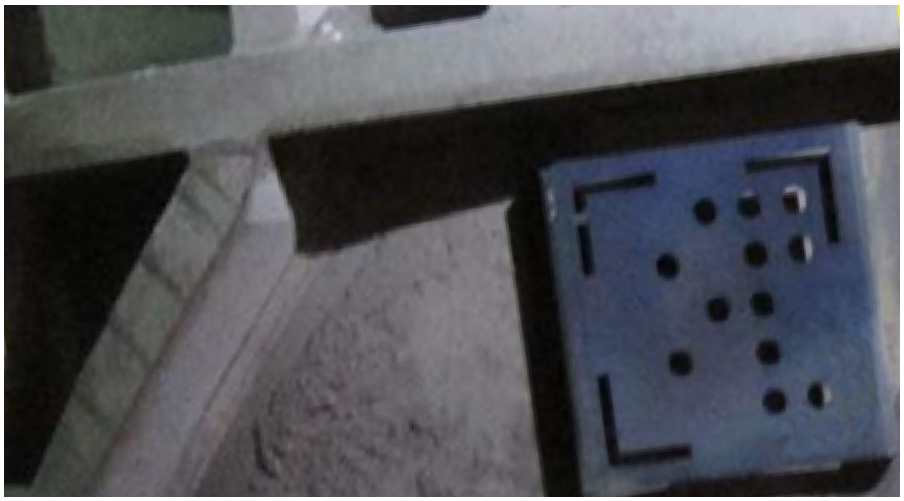

Figura 4.14: Esempio di immagine reale del terzo caso

Figura 4.15: Esempio di filtraggio completo del terzo caso

Figura 4.16: Esempio di rotazione dell'immagine del terzo caso

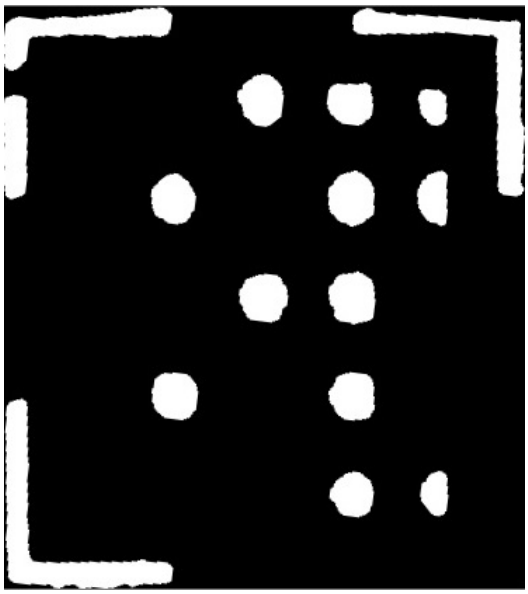

Figura 4.17: Esempio di ritaglio dell'immagine del terzo caso

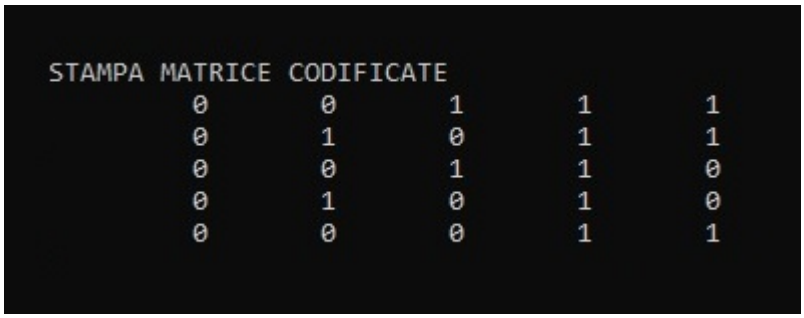

Figura 4.18: Codifica dell'immagine del terzo caso

4.2 Stress Test

Lo stress test è un procedimento adoperato per scovare delle carenze o zone critiche dell'algoritmo. Questo test consiste nell'eseguire il codice in determinate situazioni così da verificarne la correttezza, la completezza e l'affidabilità.

4.2.1 Primo stress test

Il primo collaudo effettuato riguarda l'uso di immagini peculiari, le quali presentano caratteristiche singolari come l'assenza di fori oppure dei fori presenti in una sola riga o colonna.
Questi input sono stati testati nel caso ideale dato che immagini reali con queste caratteristiche non sono disponibili. Degli esempi d'immagini con tali proprietà sono mostrate in figura 4.19, 4.21 e 4.20.
Queste immagini sono generate automaticamente da un algoritmo, illustrato nell'appendice AA.6, scritto in MATLAB. L'algoritmo riceve in input due dimensioni, M e N, con le quali genera una nuova matrice.
La nuova matrice viene riempita con dei fori situati in posizioni casuali. Una particolarità di questo algoritmo è il fatto che i cerchi non vengono disposti con i centri allineati, ma spostati con una piccola tolleranza generata casualmente.
Collaudando l'algoritmo con questi input si ottiene che comunque il codice viene eseguito fino alla fine senza errori. In particolare, supponendo di avere la figura 4.21, questa è stata generata dall'algoritmo con parametri $M = N = 6$. Alla fine del processo si ha una codifica mostrata in figura 4.22. Questa codifica risulta errata perché sono assenti tutte le righe, e/o colonne, nulle. Per la correzione di questo errore è necessario effettuare delle modifiche all'interno codice, in particolare al procedimento visto nel capitolo [2.4.2].
Le modifiche effettuate stabiliscono che dipendentemente dalla distanza tra

due fori o la distanza tra il primo, o ultimo, foro dal bordo è possibile inserire una o più righe e/o colonne completamente nulle.
Un esempio della nuova codifica dell'immagine 4.21 è mostrato in figura 4.23.

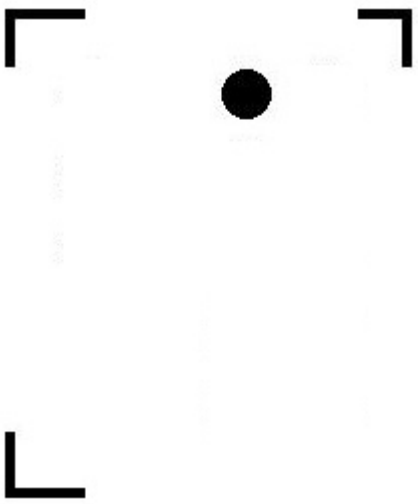

Figura 4.19: Esempio di matrice generata automaticamente: Presenta solo un foro anche se vi è parecchio spazio a disposizione, dimensioni 4×5

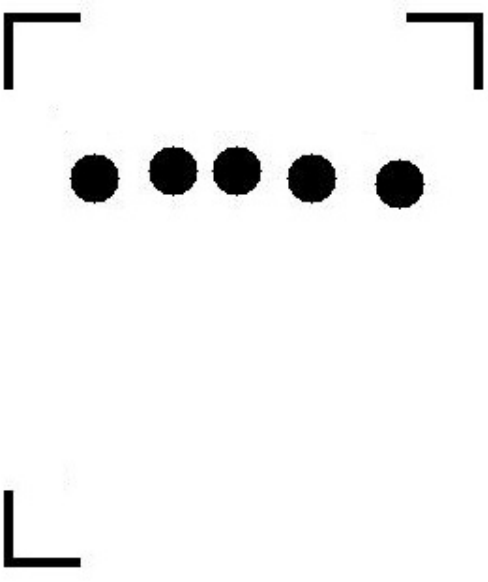

Figura 4.20: Esempio di matrice generata automaticamente: Presenta una sola riga forata completa, dimensioni $5x6$

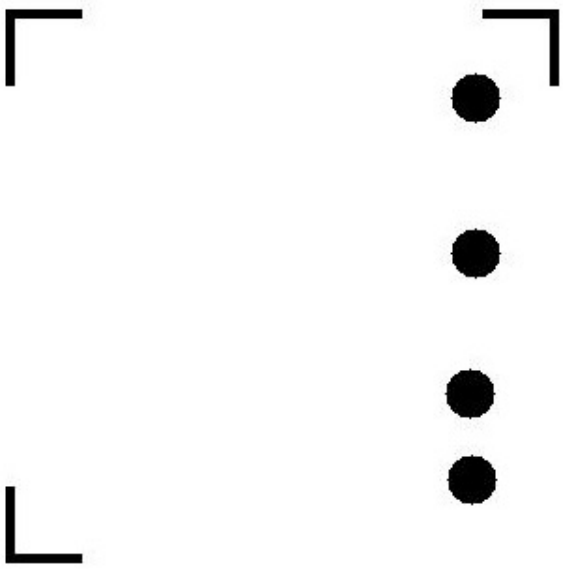

Figura 4.21: Esempio di matrice generata automaticamente: Presenta una sola riga forata completa, dimensioni $6x6$

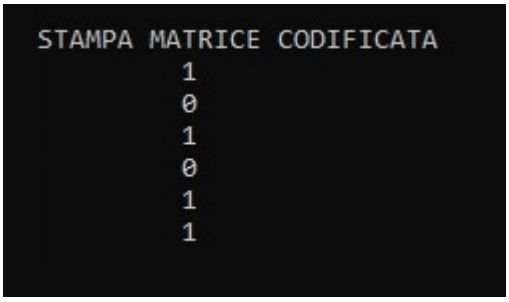

Figura 4.22: Codifica della matrice singolare: Questo tipo di codifica risulta errata perché non considera le possibili colonne vuote

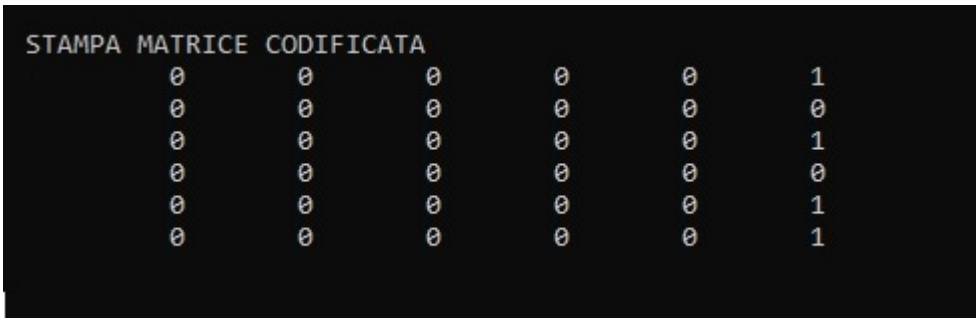

Figura 4.23: Codifica della matrice singolare: Questo tipo di codifica risulta corretta perché considera anche le possibili colonne vuote

4.2.2 Secondo stress test

La seconda tipologia di test viene effettuata mandando in esecuzione il codice per 100 volte di seguito sulla stessa immagine. Questa tipologia di test serve a determinare l'efficienza del codice, e per verificare che l'esecuzione senza pause e show di immagini intermedie possa creare dei problemi.
Inoltre, da questo test è di particolare rilevanza vedere il tempo medio di esecuzione. Il goal di progetto sarebbe una possibile esecuzione in realtime, ma purtroppo questo non è possibile quindi si cerca di avere un tempo medio relativamente basso così da permettere l'elaborazione di un maggior numero di immagini.
In particolare, tenendo molto basso il tempo di esecuzione è possibile semplicemente far scorrere la zona d'interesse davanti al sistema di acquisizioni immagini invece di dover tenere fermo il contenitore con la piastra saldata fino alla fine dell'esecuzione del programma.
Risulta, come visibile dall'immagine 4.25, che per effettuare tutti i cicli si ha un tempo di esecuzione di 42.4230 s e quindi che per ogni ciclo necessita circa di 0.4242 s.
Questo risultato è stato ottenuto su un PC che possiede un Intel® Core™ $i7-5500$ avente quattro core con un clock base di 2.4 GHz che può arrivare a 3.00 GHz. In caso di implementazioni reali in ambiente industriale o embedded [6] potrà essere necessario eseguire il codice più velocemente, allora si dovrà scegliere hardware più performante o effettuare uno studio di ottimizzazione del codice apposito non banale, dato che sicuramente le funzioni di OPENCV sono scritte in maniera efficiente. Una possibile strada potrebbe essere anche l'utilizzo di una GPU.
Un altro fattore visibile attraverso questo test è l'uso dei vari "core" del processore durante l'esecuzione del codice. Infatti, effettuando numerosi cicli è possibile monitorare le prestazioni di ogni singolo "core" della macchina sulla quale è in esecuzione del codice.
L'uso di tutti i core determina un minor tempo di esecuzione del codice.

Dalla figura 4.24 risulta proprio che l'esecuzione del codice utilizza tutti i core disponibili.

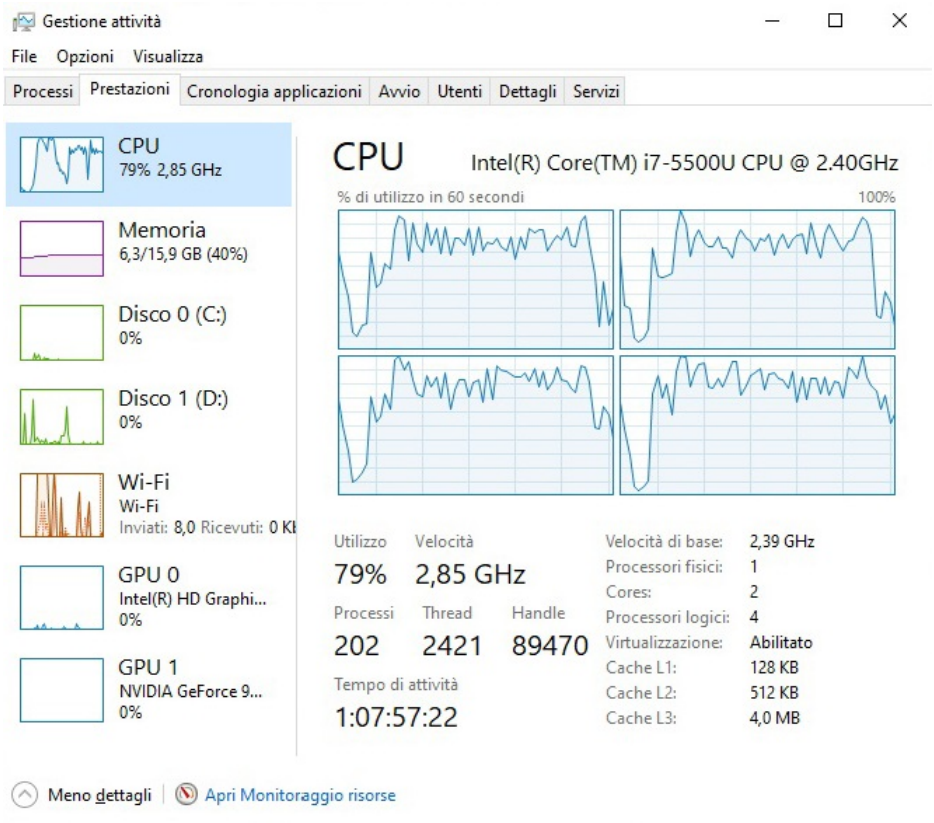

Figura 4.24: Utilizzo dei core durante l'esecuzione dell'algoritmo sotto stress

```
Il tempo di esecuzione totale e' di circa: 42.423000 Secondi

Il tempo per una singola esecuzione e' circa: 0.424230 Secondi
```

Figura 4.25: Esempio di tempo impiegato per fare 100 cicli

Conclusioni

Al termine del lavoro posso affermare che lo scopo iniziale dello sviluppo dell'algoritmo risulta raggiunto e l'implementazione della soluzione potrebbe essere realmente sfruttata nel mondo industriale per la decodifica rapida delle matrici in maniera automatica.
Dai test effettuati l'algoritmo sembra essere opportunamente robusto per svolgere il suo lavoro in maniera ottimale, anche quando l'input risulta deformato e di bassa qualità. Ovviamente per rendere più sicuro l'algoritmo è possibile implementare ulteriori controlli dei dati e dei segnali di errore.
Come ulteriore miglioramento di questo algoritmo bisognerebbe utilizzare tecniche avanzate di elaborazione delle immagini in modo da migliorare ulteriormente il filtraggio dell'immagine per adattare a più situazioni reali. Allo stesso modo è pensabile richiedere una certa accuratezza nell'installazione in maniera da lavorare in ambiente un minimo controllato (luce, dimensioni, distanze e posizionamento della telecamera).
Infine, di particolare interesse, e sicuramente una necessità per una installazione reale, sarebbe l'implementazione dell'algoritmo su una singleboard computer [1], in maniera da rendere il tutto adatto all'ambiente industriale. Ovviamente nel momento in cui si dovesse andare ad implementare su SBC industriale, sarà opportuno procedere ad una ottimizzazione del codice, cercando di controllare ed evitare ripetizioni negli accessi indicizzati agli array, in maniera tale da rendere l'esecuzione del codice più veloce possibile. A maggior ragione in macchine dotate di minor cache come SBC industriali basate su piccole architetture ARM questo ne porterà beneficio. Per quanto l'ottimizzatore del compilatore ha già fatto parte del lavoro, sicuramente un ulteriore controllo potrà dare buoni risultati.
Questa parte di lavoro non è stata presa in considerazione in fase di tesi in quanto non strettamente legata al lavoro di sviluppo sull'algoritmo.

Bibliografia

[1] William P. Birmingham, Audrey Brennan, Anurag P. Gupta, and Daniel P. Siewiorek. MICON: a single-board computer synthesis tool. *IEEE Circuits and Devices Magazine*, 4(1):37–46, 1988.

[2] Guang Deng and LW Cahill. An adaptive gaussian filter for noise reduction and edge detection. In *1993 IEEE Conference Record Nuclear Science Symposium and Medical Imaging Conference*, pages 1615–1619. IEEE, 1993. Conference Location: San Francisco, CA, USA.

[3] The Open Group. Time.h - time types, 1997. [Online; controllata il 5-Settembre-2019].

[4] Henk J. Heijmans. *Morphological image operators*, volume 4. Academic Press Boston, 1994.

[5] Intel. Opencv, Giugno 2000. [Online; controllata il 30-Luglio-2019].

[6] Peter Marwedel. *Embedded system design*, volume 1. Springer, 2006.

[7] MathWorks. MATLAB, 1984. [Online; controllata il 30-Luglio-2019].

[8] MathWorks. imadjust – mathworks, 2006. [Online; controllata il 5-Settembre-2019]".

[9] R. Medina-Carnicer, A. Carmona-Poyato, R. Muñoz-Salinas, and F. J. Madrid-Cuevas. Determining hysteresis thresholds for edge detection by combining the advantages and disadvantages of thresholding methods. *IEEE transactions on image processing*, 19(1):165–173, 2009.

[10] Kerker Milton. *The scattering of light and other electromagnetic radiation: physical chemistry: a series of monographs*, volume 16. Academic press, 2013.

[11] Gonzalez Rafael and Richard Woods. *Digital Image Processing*. Pearson Education, 2017.

[12] Sabine Süsstrunk, Robert Buckley, and Steve Swen. Standard RGB color spaces. In *Color and Imaging Conference*, volume 1999, pages 127–134. Society for Imaging Science and Technology, 1999.

[13] Wikipedia. Funzione gaussiana – wikipedia, l'enciclopedia libera, 2009. [Online; controllata il 5-Settembre-2019].

Appendice A

A.1 Algoritmo imRotate

```
Mat imRotate(Mat source, double angle) { //Funzione per la rotazione dell'immagine
        Mat dst;
        //Caso speciale in cui l'angolo sia 360.0
        if (fmod(angle, 360.0) == 0.0) //funzione che ritorna il resto tra angle e 360.0
        dst = source;
        else {
                //calcolo il centro dell'immagine
                Point2f center(source.cols / 2.0F, source.rows / 2.0F);
                //Effettuo la rotazione rispetto al centro
                Mat rot = getRotationMatrix2D(center, angle, 1.0);
                //Determino il rettangolo esterno
                Rect bbox = RotatedRect(center, source.size(),  float(angle)).boundingRect();
                //Sistemo la matrice cosi' da non tagliare nulla
                rot.at<double>(0, 2) += bbox.width / 2.0 - center.x;
                rot.at<double>(1, 2) += bbox.height / 2.0 - center.y;
                warpAffine(source, dst, rot, bbox.size(), INTER_LINEAR);
        }
        return dst;
}
```

A.2 Algoritmo Imfill

```
Mat imfill(Mat image) { //riempie i bordi dell'immagine
        //Mat fill = image.clone();
        Mat copia = image;
        for (int i = 0; i < copia.cols; i++) { //per ogni colonna
                if (copia.at<char>(0, i) == 0) {
                        floodFill(copia, Point(i, 0), 255, 0, 10, 10);
                }
                if (copia.at<char>(copia.rows - 1, i) == 0) {
                        floodFill(copia, Point(i, copia.rows - 1), 255, 0, 10, 10);
                }
        }
        copia = 255 - copia;
```

```
13              return copia;
14      }
```

A.3 Algoritmo Trova Valori

```
1       elementi trova_valori(immagine src) {
2               //funzione che distingue quali sono i cerchi e gli angoli, e i valori X/Y Max e Min
3               elementi ritorno;
4               vector<vector<Point>> angoli; //salvo gli oggetti
5               for (int i = 0; i < src.contours.size(); i++) {
6                       if (src.metric[i] > 0.10 && src.metric[i] < 0.35) {
7                               angoli.push_back(src.contours[i]); //salvo gli angoli
8                       }
9               }
10
11              int C[4];
12              int C2[4]; //corrispettive
13              C[0] = angoli[0][0].x;//-0: Cxmin
14              C[1] = angoli[0][0].y;//-1: Cymin
15              C[2] = angoli[0][0].x;//-2: Cxmax
16              C[3] = angoli[0][0].y;//-3: Cymax
17              C2[0] = angoli[0][0].y;//-0: Corrispettiva Cy a Cxmin
18              C2[1] = angoli[0][0].x;//-1: Corrispettiva Cx a Cymin
19              C2[2] = angoli[0][0].y;//-2: Corrispettiva Cy a Cxmax
20              C2[3] = angoli[0][0].x;//-3: Corrispettiva Cx a Cymax
21
22              for (int i = 0; i < angoli.size(); i++) { //determino il max, min e i corrispettivi
23                      for (int j = 0; j < angoli[i].size(); j++) {
24                              if (C[0] > angoli[i][j].x) {
25                                      C[0] = angoli[i][j].x;
26                                      C2[0] = angoli[i][j].y;
27                              }
28                              if (C[1] > angoli[i][j].y) {
29                                      C[1] = angoli[i][j].y;
30                                      C2[1] = angoli[i][j].x;
31                              }
32                              if (C[2] < angoli[i][j].x) {
33                                      C[2] = angoli[i][j].x;
34                                      C2[2] = angoli[i][j].y;
35                              }
36                              if (C[3] < angoli[i][j].y) {
37                                      C[3] = angoli[i][j].y;
38                                      C2[3] = angoli[i][j].x;
39                              }
40                      }
41              }
42
43              vector<Point2f>centri; //salvo i cerchi
44              for (int i = 0; i < src.contours.size(); i++) {
```

```
45                      if ((src.metric[i] > 0.69 && src.metric[i] < 1.0)) {
46                              centri.push_back(src.centers[i]);
47                      }
48              }
49              ritorno.Angoli = angoli;
50              ritorno.Centri = centri;
51              for (int i = 0; i < 4; i++) {
52                      ritorno.Valori[i] = C[i];
53                      ritorno.Valori2[i] = C2[i];
54              }
55              return ritorno;
56      }
```

A.4 Algoritmo Trova bordi

```
1       immagine trova_bordi(Mat src){
2               //funzione che restituisce una struttura contentente la posizione degli angoli e dei centri
3               immagine ritorno;
4               vector<vector<Point>> contorni;
5               Mat uscita = src.clone();
6               //trova i contorni degli oggetti
7               findContours(uscita, contorni, RETR_LIST, CHAIN_APPROX_NONE);
8               double b, c;
9               double metrica[100];
10              for (int i = 0; i < 100; i++) {//inizializzo l'array
11                      metrica[i] = 0;
12              }
13
14              vector<vector<Point>> contourspoly(contorni.size());
15              vector<Rect> boundRect(contorni.size());
16              vector<Point2f>center(contorni.size());
17              vector<float>radius(contorni.size());
18
19              for (size_t i = 0; i < contorni.size(); i++) { //calcolo la metrica
20                      b = contourArea(contorni[i]); //calcola l'area contenuta nel contorno
21                      c = arcLength(contorni[i], true); //calcola il perimetro del contorno
22                      metrica[i] = (4 * 3, 14 * b) / (c * c); //metrica 4*pi*Area/perimetro^2
23                      //fornisce la forma approssimata della curva
24                      approxPolyDP(contorni[i], contourspoly[i], 3, true);
25                      //fornisce centro dell'oggetto e raggio
26                      minEnclosingCircle(contourspoly[i], center[i], radius[i]);
27                      if (DEBUG) {
28                              printf("%d:␣perimetro=%f,␣area␣=%f,␣metric=%f␣\n",
29                              int(i), c, b, metrica[i]);
30                      }
31              }
32              ritorno.centers = center;
33              ritorno.contours_poly = contourspoly;
34              ritorno.contourOutput = uscita;
```

```
35              ritorno.contours = contorni;
36
37              for (int i = 0; i < 100; i++) {
38                      ritorno.metric[i] = metrica[i];
39              }
40              return ritorno;
41      }
```

A.5 Algoritmo Righe Colonne

```
1               rc righe_colonne(Mat src, elementi sorgente) {
2               //funzione che determina il valore delle righe e colonne
3               rc ritorno;
4
5               int dist;
6               if (src.rows <= src.cols) {
7                       dist = int(src.cols / 5.5);
8               }
9               else {
10                      dist = int(src.rows / 5.5);
11              }
12
13              vector<double>rg;
14              vector<double>ct;
15              for (int i = 0; i < sorgente.Centri.size(); i++) {
16                      rg.push_back(sorgente.Centri[i].y);
17                      ct.push_back(sorgente.Centri[i].x);
18              }
19
20              sort(rg.begin(), rg.end()); //ordino in maniera crescente
21              sort(ct.begin(), ct.end());
22              if (DEBUG) {
23                      printf("Stampa righe ordinate \n");
24                      for (int i = 0; i < sorgente.Centri.size(); i++) {
25                              printf("%f, ", rg[i]);
26                      }
27                      printf("\n");
28                      printf("Stampa colonne ordinate \n");
29                      for (int i = 0; i < sorgente.Centri.size(); i++) {
30                              printf("%f, ", ct[i]);
31                      }
32                      printf("\n");
33              }
34
35              vector<int> tmpr;
36              vector<int> tmpc;
37              int pr = 0, pc = 0;
38              for (int i = 0; i < ct.size(); i++) {
39                      if (i == 0) {
```

```
40                              tmpr.push_back(int(rg[i]));
41                              tmpc.push_back(int(ct[i]));
42                      }
43                      else {
44                              if (int(ct[i]) > tmpc[pc] + 10) {
45                                      tmpc.push_back(int(ct[i]));
46                                      pc = pc + 1;
47                              }
48                              if (int(rg[i]) > tmpr[pr] + 10) {
49                                      tmpr.push_back(int(rg[i]));
50                                      pr = pr + 1;
51                              }
52                      }
53              }
54              if (DEBUG) {
55                      printf("Stampa righe ordinate \n");
56                      for (int i = 0; i < tmpr.size(); i++) {
57                              printf("%d, ", tmpr[i]);
58                      }
59                      printf("\n");
60                      printf("Stampa colonne ordinate \n");
61                      for (int i = 0; i < tmpc.size(); i++) {
62                              printf("%d, ", tmpc[i]);
63                      }
64                      printf("\n");
65              }
66              if (DEBUG) printf("distanza: %d\n", dist);
67
68              vector<int>j;
69              int p = 0;
70              for (int i = 0; i < tmpr.size(); i++) {
71                      if (i == 0) {
72                              if (tmpr[i] > (dist + 10)) {
73                                      j.push_back(dist);
74                                      j.push_back(tmpr[i]);
75                                      p = p + 2;
76                              }
77                              else {
78                                      j.push_back(tmpr[i]);
79                                      p = p + 1;
80                              }
81                      }
82                      else {
83                              if ((tmpr[i] - tmpr[int(i - 1)]) > (dist + 10)) {
84                                      int t = int(tmpr[i] - tmpr[int(i - 1)]) / (dist + 10);
85                                      for (int k = p; k < p + t; k++) {
86                                              j.push_back(j[int(k - 1)] + dist);
87                                      }
88                                      p = p + t;
```

```
89                                  }
90                                  else {
91                                          j.push_back(tmpr[i]);
92                                          p = p + 1;
93                                  }
94                          }
95                  }
96                  tmpr = j;
97                  if (src.rows - tmpr[tmpr.size() - 1] > dist + 10) {
98                          int t = int((src.rows - tmpr[tmpr.size()]) / dist);
99                          t = t - 1;
100                         for (int k = p; k < p + t; k++) {
101                                 tmpr.push_back(tmpr[int(k - 1)] + dist);
102                         }
103                 }
104                 vector<int>l;
105                 p = 0;
106                 for (int i = 0; i < tmpc.size(); i++) {
107                         if (i == 0) {
108                                 if (tmpc[i] > (dist + 10)) {
109                                         l.push_back(dist);
110                                         l.push_back(tmpc[i]);
111                                         p = p + 2;
112                                 }
113                                 else {
114                                         l.push_back(tmpc[i]);
115                                         p = p + 1;
116                                 }
117                         }
118                         else {
119                                 if ((tmpc[i] - tmpc[int(i - 1)]) > (dist + 10)) {
120                                         int t = int(tmpc[i] - tmpc[int(i - 1)]) / (dist + 10);
121                                         for (int k = p; k < p + t; k++) {
122                                                 l.push_back(l[int(k - 1)] + dist);
123                                         }
124                                         p = p + t;
125                                 }
126                                 else {
127                                         l.push_back(tmpc[i]);
128                                         p = p + 1;
129                                 }
130                         }
131                 }
132                 tmpc = l;
133                 if (src.cols - tmpc[tmpc.size() - 1] > dist + 10) {
134                         int t = int((src.cols - tmpc[tmpc.size()]) / dist);
135                         t = t - 1;
136                         for (int k = p; k < p + t; k++) {
137                                 tmpc.push_back(tmpc[int(k - 1)] + dist);
```

```
138                     }
139             }
140             if (DEBUG) {
141                     printf("Stampa righe complete \n");
142                     for (int i = 0; i < tmpr.size(); i++) {
143                             printf("%d, ", tmpr[i]);
144                     }
145                     printf("\n");
146                     printf("Stampa colonne complete \n");
147                     for (int i = 0; i < tmpc.size(); i++) {
148                             printf("%d, ", tmpc[i]);
149                     }
150                     printf("\n");
151             }
152             //ritorno solo le colonne singolari, stessa cosa per le righe
153             ritorno.cl = tmpc;
154             ritorno.rg = tmpr;
155             return ritorno;
156     }
```

A.6 Algoritmo generatore di matrici

```
1
2       x=input('Inserire numero di colonne: ');
3       y=input('Inserire numero di righe: ');;
4       str='immaginecreata.bmp';
5       distanzab_cerchio=45;%distanza dal bordo
6       raggio=13; %raggio di ogni cerchio
7       raggior=raggio+0.5;
8       distanzacc=40;%distanza centro-centro tra due fori
9       distanzab_angolo=floor(distanzab_cerchio/4);%distanza angoli
10      spessore=floor(distanzab_angolo/2);%spessore degli angoli
11
12      %calcolo le dimensioni in pixel dipendentemente dalle specifiche fornite
13      m=distanzab_cerchio*2+(raggior*2)*floor(y)+
14              (distanzacc-raggior*2)*(floor(y)-1);
15      n=distanzab_cerchio*2+(raggior*2)*floor(x)+
16              (distanzacc-raggior*2)*(floor(x)-1);
17      matrice = zeros(m,n);
18
19      %inserisco gli angoli
20      pos=distanzab_angolo;
21      for j=1:spessore
22              for i=distanzab_angolo:(distanzacc+distanzab_angolo)
23                      matrice(pos,i)=1;
24                      matrice(i, pos)=1;
25                      matrice(i, n-pos)=1;
26                      matrice(m-pos,i)=1;
27                      matrice(pos,n-i)=1;
```

```
28                      matrice(m-i, pos)=1;
29              end
30              pos=pos+1;
31      end
32
33      %genero numeri casuali con valori 0 o 1.
34      %numero e' una matrice X x Y con la posizione dei cerchi.
35      posizione_cerchi=randi([0 1], y, x);
36      posizione=[];
37      posr=distanzab_cerchio+raggio+1;
38      posc=distanzab_cerchio+raggio+1;
39      p=1;
40      for i=1:y %righe
41              for j=1:x %colonne
42                      if(posizione_cerchi(i,j)==1)
43                              posizione(p,1)=posr+randi([-(distanzacc/10)
44                               (distanzacc/10)], 1, 1);
45                              posizione(p,2)=posc+randi([-(distanzacc/10)
46                               (distanzacc/10)], 1, 1);
47                              p=p+1;
48                      end
49                      posc=posc+distanzacc;
50              end
51              posr=posr+distanzacc;
52              posc=distanzab_cerchio+raggio+1;%resettare la riga
53      end
54
55      %disegno la matrice inserendo i cerchi
56      [x,y] = meshgrid(1:n, 1:m);
57      for i=1:(p-1)
58              inside = (x - posizione(i,2)).^2 + (y - posizione(i,1)).^2 <=
59               raggio^2; %genero i cerchi nella posizione desiderata
60              %sommo la matrice di base con la matrice contenente solo il cerchio
61              matrice=matrice+inside;
62      end
63      %salvo la matrice nella posizione contenuta in str
64      imwrite(matrice, str);
65      %stampo su schermo la matrice
66      figure; imshow(matrice);
```

Appendice B

B.1 Algoritmo completo

```
1   //includo le varie librerie
2   #include<opencv2/opencv.hpp>
3   #include<opencv2/imgproc/imgproc.hpp>
4   #include<opencv2/core/core.hpp>
5   #include<opencv2/highgui/highgui.hpp>
6   #include<iostream> //standard
7   #include<time.h> //per il get time
8   #include<string.h> //per convertire i numeri in stringa
9   #include<algorithm> //per utilizzare il sort
10  #include<math.h> //libreria con funzioni matematiche
11
12  #define DEBUG 0
13  #define LOOP 1
14
15  using namespace std;
16  using namespace cv;
17
18
19  struct rc {
20  //struttura contenente i vettori delle righe e delle colonne
21  vector<int> rg;
22  vector<int> cl;
23  };
24
25  struct immagine {
26  //struttura contentente l'immagine, i contorni, i contorni
27  approssimati e i valori metrici
28  vector<vector<Point>> contours; //salvo gli oggetti
29  vector<vector<Point>> contours_poly; //salvo gli oggetti approssimati
30  vector<Point2f> centers; //salvo i centri
31  Mat contourOutput; //salvo l'immagine
32  double metric[100]; //salvo il valore metric degli oggetti
33  };
34
```

```
35
36          struct elementi {
37                  //struttura contentente la posizione degli angoli, posizione
38                   centri, Valore massimi e minimi X/Y e corrispettivi Y/X
39                  vector<vector<Point>> Angoli;
40                  vector<Point2f> Centri;
41                  int Valori[4];
42                  int Valori2[4];
43          };
44
45          Mat imRotate(Mat source, double angle) { //Funzione per la rotazione dell'immagine
46                  Mat dst;
47                  //Caso speciale in cui l'angolo sia 360.0
48                  if (fmod(angle, 360.0) == 0.0) //funzione che ritorna il resto tra angle e 360.0
49                  dst = source;
50                  else {
51                          //calcolo il centro dell'immagine
52                          Point2f center(source.cols / 2.0F, source.rows / 2.0F);
53                          //Effettuo la rotazione rispetto al centro
54                          Mat rot = getRotationMatrix2D(center, angle, 1.0);
55                          //Determino il rettangolo esterno
56                          Rect bbox = RotatedRect(center, source.size(),  float(angle)).boundingRect();
57                          //Sistemo la matrice cosi' da non tagliare nulla
58                          rot.at<double>(0, 2) += bbox.width / 2.0 - center.x;
59                          rot.at<double>(1, 2) += bbox.height / 2.0 - center.y;
60                          warpAffine(source, dst, rot, bbox.size(), INTER_LINEAR);
61                  }
62                  return dst;
63          }
64
65          Mat im2bw(Mat image) { //Trovo i bordi dell'immagine con una soglia di 210 su 255
66                  Mat im = image.clone();
67                  threshold(image, im, 210, 255, THRESH_BINARY);
68                  return im;
69          }
70
71
72          Mat imfill(Mat image) { //riempie i bordi dell'immagine
73                  //Mat fill = image.clone();
74                  Mat copia = image;
75                  for (int i = 0; i < copia.cols; i++) { //per ogni colonna
76                          if (copia.at<char>(0, i) == 0) {
77                                  floodFill(copia, Point(i, 0), 255, 0, 10, 10);
78                          }
79                          if (copia.at<char>(copia.rows - 1, i) == 0) {
80                                  floodFill(copia, Point(i, copia.rows - 1), 255, 0, 10, 10);
81                          }
82                  }
83                  copia = 255 - copia;
```

```
84              return copia;
85      }
86
87      immagine trova_bordi(Mat src){
88              //funzione che restituisce una struttura contentente la posizione degli angoli e dei centri
89              immagine ritorno;
90              vector<vector<Point>> contorni;
91              Mat uscita = src.clone();
92              //trova i contorni degli oggetti
93              findContours(uscita, contorni, RETR_LIST, CHAIN_APPROX_NONE);
94              double b, c;
95              double metrica[100];
96              for (int i = 0; i < 100; i++) {//inizializzo l'array
97                      metrica[i] = 0;
98              }
99
100             vector<vector<Point>> contourspoly(contorni.size());
101             vector<Rect> boundRect(contorni.size());
102             vector<Point2f>center(contorni.size());
103             vector<float>radius(contorni.size());
104
105             for (size_t i = 0; i < contorni.size(); i++) { //calcolo la metrica
106                     b = contourArea(contorni[i]); //calcola l'area contenuta nel contorno
107                     c = arcLength(contorni[i], true); //calcola il perimetro del contorno
108                     metrica[i] = (4 * 3, 14 * b) / (c * c); //metrica 4*pi*Area/perimetro^2
109                     //fornisce la forma approssimata della curva
110                     approxPolyDP(contorni[i], contourspoly[i], 3, true);
111                     //fornisce centro dell'oggetto e raggio
112                     minEnclosingCircle(contourspoly[i], center[i], radius[i]);
113                     if (DEBUG) {
114                             printf("%d:␣perimetro=%f,␣area␣=%f,␣metric=%f␣\n",
115                              int(i), c, b, metrica[i]);
116                     }
117             }
118             ritorno.centers = center;
119             ritorno.contours_poly = contourspoly;
120             ritorno.contourOutput = uscita;
121             ritorno.contours = contorni;
122
123             for (int i = 0; i < 100; i++) {
124                     ritorno.metric[i] = metrica[i];
125             }
126             return ritorno;
127     }
128
129     elementi trova_valori(immagine src) {
130             //funzione che distingue quali sono i cerchi e gli angoli, e i valori X/Y Max e Min
131             elementi ritorno;
132             vector<vector<Point>> angoli; //salvo gli oggetti
```

```
133             for (int i = 0; i < src.contours.size(); i++) {
134                     if (src.metric[i] > 0.10 && src.metric[i] < 0.35) {
135                             angoli.push_back(src.contours[i]); //salvo gli angoli
136                     }
137             }
138
139             int C[4];
140             int C2[4]; //corrispettive
141             C[0] = angoli[0][0].x;//-0: Cxmin
142             C[1] = angoli[0][0].y;//-1: Cymin
143             C[2] = angoli[0][0].x;//-2: Cxmax
144             C[3] = angoli[0][0].y;//-3: Cymax
145             C2[0] = angoli[0][0].y;//-0: Corrispettiva Cy a Cxmin
146             C2[1] = angoli[0][0].x;//-1: Corrispettiva Cx a Cymin
147             C2[2] = angoli[0][0].y;//-2: Corrispettiva Cy a Cxmax
148             C2[3] = angoli[0][0].x;//-3: Corrispettiva Cx a Cymax
149
150             for (int i = 0; i < angoli.size(); i++) { //determino il max, min e i corrispettivi
151                     for (int j = 0; j < angoli[i].size(); j++) {
152                             if (C[0] > angoli[i][j].x) {
153                                     C[0] = angoli[i][j].x;
154                                     C2[0] = angoli[i][j].y;
155                             }
156                             if (C[1] > angoli[i][j].y) {
157                                     C[1] = angoli[i][j].y;
158                                     C2[1] = angoli[i][j].x;
159                             }
160                             if (C[2] < angoli[i][j].x) {
161                                     C[2] = angoli[i][j].x;
162                                     C2[2] = angoli[i][j].y;
163                             }
164                             if (C[3] < angoli[i][j].y) {
165                                     C[3] = angoli[i][j].y;
166                                     C2[3] = angoli[i][j].x;
167                             }
168                     }
169             }
170
171             vector<Point2f>centri; //salvo i cerchi
172             for (int i = 0; i < src.contours.size(); i++) {
173                     if ((src.metric[i] > 0.69 && src.metric[i] < 1.0)) {
174                             centri.push_back(src.centers[i]);
175                     }
176             }
177             ritorno.Angoli = angoli;
178             ritorno.Centri = centri;
179             for (int i = 0; i < 4; i++) {
180                     ritorno.Valori[i] = C[i];
181                     ritorno.Valori2[i] = C2[i];
```

```
182             }
183             return ritorno;
184     }
185
186     rc righe_colonne(Mat src, elementi sorgente) {
187             //funzione che determina il valore delle righe e colonne
188             rc ritorno;
189
190             int dist;
191             if (src.rows <= src.cols) {
192                     dist = int(src.cols / 5.5);
193             }
194             else {
195                     dist = int(src.rows / 5.5);
196             }
197
198             vector<double>rg;
199             vector<double>ct;
200             for (int i = 0; i < sorgente.Centri.size(); i++) {
201                     rg.push_back(sorgente.Centri[i].y);
202                     ct.push_back(sorgente.Centri[i].x);
203             }
204
205             sort(rg.begin(), rg.end()); //ordino in maniera crescente
206             sort(ct.begin(), ct.end());
207             if (DEBUG) {
208                     printf("Stampa righe ordinate \n");
209                     for (int i = 0; i < sorgente.Centri.size(); i++) {
210                             printf("%f, ", rg[i]);
211                     }
212                     printf("\n");
213                     printf("Stampa colonne ordinate \n");
214                     for (int i = 0; i < sorgente.Centri.size(); i++) {
215                             printf("%f, ", ct[i]);
216                     }
217                     printf("\n");
218             }
219
220             vector<int> tmpr;
221             vector<int> tmpc;
222             int pr = 0, pc = 0;
223             for (int i = 0; i < ct.size(); i++) {
224                     if (i == 0) {
225                             tmpr.push_back(int(rg[i]));
226                             tmpc.push_back(int(ct[i]));
227                     }
228                     else {
229                             if (int(ct[i]) > tmpc[pc] + 10) {
230                                     tmpc.push_back(int(ct[i]));
```

```
231                                     pc = pc + 1;
232                             }
233                             if (int(rg[i]) > tmpr[pr] + 10) {
234                                     tmpr.push_back(int(rg[i]));
235                                     pr = pr + 1;
236                             }
237                     }
238             }
239             if (DEBUG) {
240                     printf("Stampa righe ordinate \n");
241                     for (int i = 0; i < tmpr.size(); i++) {
242                             printf("%d, ", tmpr[i]);
243                     }
244                     printf("\n");
245                     printf("Stampa colonne ordinate \n");
246                     for (int i = 0; i < tmpc.size(); i++) {
247                             printf("%d, ", tmpc[i]);
248                     }
249                     printf("\n");
250             }
251             if (DEBUG) printf("distanza: %d\n", dist);
252
253             vector<int>j;
254             int p = 0;
255             for (int i = 0; i < tmpr.size(); i++) {
256                     if (i == 0) {
257                             if (tmpr[i] > (dist + 10)) {
258                                     j.push_back(dist);
259                                     j.push_back(tmpr[i]);
260                                     p = p + 2;
261                             }
262                             else {
263                                     j.push_back(tmpr[i]);
264                                     p = p + 1;
265                             }
266                     }
267                     else {
268                             if ((tmpr[i] - tmpr[int(i - 1)]) > (dist + 10)) {
269                                     int t = int(tmpr[i] - tmpr[int(i - 1)]) / (dist + 10);
270                                     for (int k = p; k < p + t; k++) {
271                                             j.push_back(j[int(k - 1)] + dist);
272                                     }
273                                     p = p + t;
274                             }
275                             else {
276                                     j.push_back(tmpr[i]);
277                                     p = p + 1;
278                             }
279                     }
```

```
280                 }
281                 tmpr = j;
282                 if (src.rows - tmpr[tmpr.size() - 1] > dist + 10) {
283                         int t = int((src.rows - tmpr[tmpr.size()]) / dist);
284                         t = t - 1;
285                         for (int k = p; k < p + t; k++) {
286                                 tmpr.push_back(tmpr[int(k - 1)] + dist);
287                         }
288                 }
289                 vector<int>l;
290                 p = 0;
291                 for (int i = 0; i < tmpc.size(); i++) {
292                         if (i == 0) {
293                                 if (tmpc[i] > (dist + 10)) {
294                                         l.push_back(dist);
295                                         l.push_back(tmpc[i]);
296                                         p = p + 2;
297                                 }
298                                 else {
299                                         l.push_back(tmpc[i]);
300                                         p = p + 1;
301                                 }
302                         }
303                         else {
304                                 if ((tmpc[i] - tmpc[int(i - 1)]) > (dist + 10)) {
305                                         int t = int(tmpc[i] - tmpc[int(i - 1)]) / (dist + 10);
306                                         for (int k = p; k < p + t; k++) {
307                                                 l.push_back(l[int(k - 1)] + dist);
308                                         }
309                                         p = p + t;
310                                 }
311                                 else {
312                                         l.push_back(tmpc[i]);
313                                         p = p + 1;
314                                 }
315                         }
316                 }
317                 tmpc = l;
318                 if (src.cols - tmpc[tmpc.size() - 1] > dist + 10) {
319                         int t = int((src.cols - tmpc[tmpc.size()]) / dist);
320                         t = t - 1;
321                         for (int k = p; k < p + t; k++) {
322                                 tmpc.push_back(tmpc[int(k - 1)] + dist);
323                         }
324                 }
325                 if (DEBUG) {
326                         printf("Stampa righe complete \n");
327                         for (int i = 0; i < tmpr.size(); i++) {
328                                 printf("%d, ", tmpr[i]);
```

```
329                     }
330                     printf("\n");
331                     printf("Stampa colonne complete \n");
332                     for (int i = 0; i < tmpc.size(); i++) {
333                             printf("%d, ", tmpc[i]);
334                     }
335                     printf("\n");
336             }
337             //ritorno solo le colonne singolari, stessa cosa per le righe
338             ritorno.cl = tmpc;
339             ritorno.rg = tmpr;
340             return ritorno;
341     }
342
343     int main() {
344             clock_t start, end; //definisco il tempo
345             double tempo;
346             start = clock(); //inizio a contare
347             //imread funzione per leggere l'immagine, questa puo' avere due parametri
348             //-nome o path immagine
349             //-numero intero che di default e' 1, ma se si inserisce 0 allora
350             // la lettura avviene gia' in bianco e nero
351
352             Mat img = imread("./Immagini/posizione6.bmp", 1); //carico l'immagine
353             if (img.data == NULL) { //se non riesce a leggere nulla
354                     printf("Errore nell'apertura dell'immagine \n");
355                     return -1;
356             }
357             if (DEBUG){
358                     printf("Dimensioni immagini: colonne=%d e righe=%d\n", img.cols, img.rows);
359             }
360
361             if (DEBUG) {//stampo l'immagine
362                     namedWindow("Immagine Originale", WINDOW_AUTOSIZE);
363                     imshow("Immagine Originale", img);
364                     waitKey(0);
365             }
366
367             int max = 1;
368             if (LOOP) {
369                     max = 100;
370             }
371             for (int k = 0; k < max; k++) {//controllo se voglio un loop (fase test)
372                     //converto l'immagine in scala di grigi
373                     Mat imgGray = img.clone();
374                     cvtColor(img, imgGray, COLOR_BGR2GRAY); //scala di grigi
375                     //Filtro gaussiano per la sfocatura
376                     GaussianBlur(imgGray, imgGray, Size(3, 3), 0, 0, BORDER_DEFAULT);
377                     if (DEBUG) {
```

```
378                 imshow("Immagine Grigio", imgGray);
379                 waitKey(0);
380         }
381
382         imgGray = 255 - imgGray; //negativo
383         if (DEBUG) {
384                 imshow("Immagine Negativo", imgGray);
385                 waitKey(0);
386         }
387
388         imgGray = im2bw(imgGray); //torno l'immagine con 255 nei valori sopra il 210
389         if (DEBUG) {
390                 imshow("Immagine Campionata", imgGray);
391                 waitKey(0);
392         }
393
394         Canny(imgGray, imgGray, 50, 200, 3, false); //applico filtro di Canny
395
396         if (DEBUG) {
397                 imshow("Immagine Canny", imgGray);
398                 waitKey(0);
399         }
400
401         imgGray = imfill(imgGray); //riempio i buchi
402         if (DEBUG) {
403                 imshow("Immagine Riempita", imgGray);
404                 waitKey(0);
405         }
406
407         int erosione_size = 3; //erodo l'immagine per eliminare le imperfezioni
408         Mat element = getStructuringElement(MORPH_ELLIPSE,
409         Size(2 * erosione_size + 1, 2 * erosione_size + 1),
410          Point(erosione_size, erosione_size));
411         erode(imgGray, imgGray, element);
412         if (DEBUG) {
413                 imshow("Immagine Erosa", imgGray);
414                 waitKey(0);
415         }
416
417         int dilation_size = 3; //dilato l'immagine ripulita
418         element = getStructuringElement(MORPH_ELLIPSE,
419         Size(2 * dilation_size + 1, 2 * dilation_size + 1),
420          Point(dilation_size, dilation_size));
421         dilate(imgGray, imgGray, element);
422         if (DEBUG) {
423                 imshow("Immagine Dilatata", imgGray);
424                 waitKey(0);
425         }
426         immagine iniziale = trova_bordi(imgGray);
```

```
427                 Mat drawing = Mat::zeros(imgGray.size(), CV_8UC3);
428                 if (DEBUG > 1) {
429                         for (size_t i = 0; i < iniziale.contours.size(); i++) {
430                                 Scalar color = Scalar(150, 150, 150);
431                                 drawContours(drawing, iniziale.contours_poly, (int)i, color);
432                                 circle(drawing, iniziale.centers[i], 3, color, 2);
433                                 putText(drawing, to_string(iniziale.metric[i]),
434                                  iniziale.centers[i], FONT_HERSHEY_SIMPLEX, 0.4,
435                                   Scalar(0, 143, 143), 2);
436                         }
437                         imshow("Contours", drawing);
438                         waitKey(0);
439                 }
440                 elementi primo = trova_valori(iniziale);
441
442                 double y = abs(primo.Valori[1] - primo.Valori2[0]);
443                 double x = abs(primo.Valori[2] - primo.Valori[0]);
444
445                 double angle = atan2(y, x);
446                 double degree = angle * 180 / CV_PI;
447                 if (degree < 5.0) {
448                         degree = 0.0;
449                 }
450                 if (DEBUG) {
451                         printf("y=%f, x=%f \n", y, x);
452                         printf("Angolo = %f\n", degree);
453                 }
454
455                 Mat rotate = imRotate(iniziale.contourOutput, -degree);
456                 iniziale = trova_bordi(rotate);
457
458                 elementi secondo = trova_valori(iniziale);
459
460                 Rect myR(secondo.Valori[0], secondo.Valori[1],
461                  secondo.Valori[2] - secondo.Valori[0], secondo.Valori[3] - secondo.Valori[1]);
462                 Mat imgCroped = rotate(myR);
463                 if (DEBUG) {
464                         namedWindow("Immagine ritagliata", WINDOW_AUTOSIZE);
465                         imshow("Immagine ritagliata", imgCroped);
466                         waitKey(0);
467                 }
468
469                 immagine finale = trova_bordi(imgCroped);
470
471
472                 drawing = Mat::zeros(imgCroped.size(), CV_8UC3);
473                 if (DEBUG > 1) {
474                         for (size_t i = 0; i < finale.contours.size(); i++) {
475                                 Scalar color = Scalar(150, 150, 150);
```

```
476                         drawContours(drawing, finale.contours_poly,
477                                 (int)i, color);
478                         circle(drawing, finale.centers[i], 3, color, 2);
479                         putText(drawing, to_string(finale.metric[i]),
480                                 finale.centers[i],FONT_HERSHEY_SIMPLEX, 0.4,
481                                 Scalar(0, 143, 143), 2);
482                 }
483                 imshow("Contours", drawing);
484                 waitKey(0);
485         }
486         elementi terzo = trova_valori(finale);
487         if (DEBUG > 1) {
488                 drawing = Mat::zeros(imgCroped.size(), CV_8UC3);
489                 for (size_t i = 0; i < finale.contours.size(); i++) {
490                         drawContours(drawing, finale.contours_poly,
491                                 (int)i, Scalar(255, 255, 255));
492                 }
493                 for (size_t i = 0; i < terzo.Centri.size(); i++) {
494                         Scalar color = Scalar(150, 0, 0);
495                         circle(drawing, terzo.Centri[i], 3, color, 2);
496                 }
497                 imshow("Stampo solo i cerchi", drawing);
498                 waitKey(0);
499         }
500
501
502         rc valori = righe_colonne(imgCroped, terzo);
503
504         vector <vector<int>> A(valori.rg.size());
505         for (int i = 0; i < valori.rg.size(); i++) {
506                 A[i] = vector<int>(valori.cl.size());
507         }
508
509         for (int i = 0; i < valori.rg.size(); i++) {
510                 for (int k = 0; k < valori.cl.size(); k++) {
511                         A[i][k] = 0;
512                 }
513         }
514
515         int tr2 = 0, tc2 = 0;
516         int n = 1;
517         for (int i = 0; i < terzo.Centri.size(); i++) {
518                 tr2 = tc2 = 0;
519                 for (int k = 0; k < valori.rg.size() && n; k++) {
520                         double tmp1 = double(valori.rg[k] - 10);
521                         double tmp2 = double(valori.rg[k] + 10);
522                         if (tmp1 <= terzo.Centri[i].y && terzo.Centri[i].y <= tmp2) {
523                                 tr2 = k;
524                                 n = 0;
```

```
525                                     }
526                                 }
527                                 n = 1;
528                                 for (int k = 0; k < valori.cl.size() && n; k++) {
529                                     double tmp1 = double(valori.cl[k] - 10);
530                                     double tmp2 = double(valori.cl[k] + 10);
531                                     if (tmp1 <= terzo.Centri[i].x && terzo.Centri[i].x <= tmp2) {
532                                         tc2 = k;
533                                         n = 0;
534                                     }
535                                 }
536                                 if (DEBUG) printf("%d e %d \n", tr2, tc2);
537                                 A[tr2][tc2] = 1;
538                                 n = 1;
539                             }
540
541                             if (DEBUG) {
542                                 printf("\nSTAMPA MATRICE CODIFICATE \n");
543
544                                 for (int i = 0; i < valori.rg.size(); i++) {
545                                     for (int k = 0; k < valori.cl.size(); k++) {
546                                         printf("\t %d", A[i][k]);
547                                     }
548                                     printf("\n");
549                                 }
550                             }
551                         }
552                         end = clock();
553                         tempo = ((double)(end - start)) / CLOCKS_PER_SEC;
554                         printf("\nIl tempo di esecuzione totale e' di circa: %f Secondi\n", tempo);
555                         printf("\n");
556                         printf("\n Il tempo per una singola esecuzione e' circa: %f Secondi\n", tempo / 100);
557                         return 0;
558                 }
```

Ringraziamenti

Ringrazio il mio relatore, il Prof. Sergio Carrato, per il tempo dedicatomi e l'entusiasmo trasmesso verso questo progetto.
Ringrazio il mio correlatore, Ing. Piergiorgio Menia, per la disponibilità dimostrata, gli aiuti e i consigli fornitomi ed è solo merito suo se questo progetto ha preso vita.
Il ringraziamento più grande va a mio padre e mia madre, perché senza i loro sacrifici e il loro sostegno non sarei qui. Un pezzo di questo risultato è loro.
Ringrazio i miei fratelli, Stefano e Lisa, per l'appoggio fornitomi durante questi anni.
Ringrazio i miei nonni, Giuseppe, Serafina, Giovanna e Rosario che sono stati sempre presenti nei giorni difficili.
Ringrazio mio zio, Gianni Licitra, che ha creduto in me sin dall'inizio di questa avventura ed è stato uno dei miei più grandi sostenitori.
Ringrazio i miei migliori amici, Lorenzo Chessari e Carmelo Distefano, che nonostante i chilometri che ci separano sono stati sempre presenti supportando le mie scelte e strappandomi sempre un sorriso.
Ringrazio la mia fidanzata per il sostegno, ma anche perché mi ha sopportato in questi mesi.
Ringrazio i nuovi amici trovati a Trieste, con cui ho affrontato l'università, rendendo tutto ciò una bellissima esperienza da ricordare.
E, infine, ringrazio tutti coloro che in qualche modo hanno contribuito al raggiungimento di questo obiettivo.