

UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II
SCUOLA POLITECNICA E DELLE SCIENZE DI BASE
DIPARTIMENTO DI INGEGNERIA ELETTRICA E TECNOLOGIE DELL'INFORMAZIONE



CORSO DI LAUREA MAGISTRALE IN INFORMATICA

Edge Detection Evaluation

Progetto per il corso di Visione Computazionale

Docente

Prof. Francesco Isgrò

Studente

Andrea Santangelo

N97000291

Anno Accademico 2019 - 2020

Indice

Elenco delle figure	iii
1 Introduzione	1
2 Algoritmi	2
2.1 Search-based algorithm (Gradiente)	2
2.1.1 Operatori per il calcolo del gradiente	3
2.1.1.1 Operatore di Roberts Cross	4
2.1.1.2 Operatore di Sobel	6
2.1.1.3 Operatore di Prewitt	8
2.1.2 Algoritmo mono fase	10
2.1.2.1 Gaussian Blur	10
2.1.2.2 Gradiente	11
2.1.2.3 Treshold	12
2.1.3 Algoritmo multi fase (Canny)	15
2.1.3.1 Edge Enhancement	15
Smoothing	15
Gradiente	15
2.1.3.2 Non-Maximum Suppression	16
2.1.3.3 Hysteresis Thresholding	17
Double Thesholding	17
Edge tracking by hysteresis	19
2.2 Zero-crossing algorithm (Derivata seconda)	21
2.2.1 Marr–Hildreth	21
2.2.1.1 Laplacian of Gaussian (LoG)	22
2.2.1.2 Zero-Crossing e Thresholding	23

3 Metriche	26
3.1 Mean Absolute Error (MAE)	26
3.2 Pratt's Figure of Merit (PFOM)	27
3.3 Map Quality (MQ)	28
4 Analisi dei risultati	29
Appendices	44
A Applicativo	45
A.1 Design	46
A.2 Funzioni principali	47
A.2.1 ImageUtil	47
A.2.2 SinglePhaseEdgeDetector	48
A.2.3 MultiPhaseEdgeDetection	48
A.2.4 ZeroCrossingEdgeDetection	49
A.2.5 MetricsFunction	50
A.3 Contenuto archivio	50
A.4 Istruzioni d'uso	51
A.4.1 Prerequisiti	52
A.4.2 Installazione e configurazione dell'ambiente	53
A.4.3 Visualizzazione degli step per ogni algoritmo	54
A.4.4 Benchmark	56

Elenco delle figure

2.1	Immagine originale usata per gli operatori	3
2.2	Roberts Cross Masks	4
2.3	Convoluzione tramite maschera orizzontale di Roberts Cross	5
2.4	Convoluzione tramite maschera verticale di Roberts Cross	5
2.5	Gradiente ottenuto dall'applicazione dell'operatore di Roberts Cross	5
2.6	Sobel Masks	6
2.7	Convoluzione tramite maschera orizzontale di Sobel	6
2.8	Convoluzione tramite maschera verticale di Sobel	6
2.9	Gradiente ottenuto dall'applicazione dell'operatore di Sobel	7
2.10	Prewitt Masks	8
2.11	Convoluzione tramite maschera orizzontale di Prewitt	8
2.12	Convoluzione tramite maschera verticale di Prewitt	8
2.13	Gradiente ottenuto dall'applicazione dell'operatore di Prewitt	9
2.14	Applicazione di un filtro gaussiano con $\sigma = 3$ e dimensione = 15	11
2.15	Gradiente ottenuto con Roberts Cross - Sobel - Prewitt	11
2.16	Gradiente ottenuto con Roberts Cross - Sobel - Prewitt (Blurring)	12
2.17	Applicazione del threshold (= 80) ai gradienti di Roberts Cross - Sobel - Prewitt	13
2.18	Applicazione del threshold (= 80) ai gradienti di Roberts Cross - Sobel - Prewitt (Blurring)	13
2.19	Step eseguiti da un algoritmo "Mono fase" con operatore di Sobel	14
2.20	Non-Maximum Suppression Roberts Cross - Sobel - Prewitt	17
2.21	Double Thresholding Roberts Cross - Sobel - Prewitt	19
2.22	Meccanismo Hysteresis	19
2.23	Edge tracking by hysteresis Roberts Cross - Sobel - Prewitt	20
2.24	Step eseguiti da un algoritmo "Multi fase" (Canny) con operatore di Sobel	20

2.25	Maschere laplaciane	21
2.26	Applicazione filtro ”Laplacian of Gaussian (LoG)” ($\sigma = 1.6$, $dim = 5x5$)	22
2.27	Applicazione algoritmo di Zero-Crossing	23
2.28	Applicazione algoritmo di Zero-Crossing con thresholding	24
2.29	Step eseguiti da un algoritmo ”Zero-Crossing” (MarrHildreth)	25
4.1	Immagine originale - scala di grigi - GroundTruth 2018.jpg	31
4.2	Benchmark algoritmi ”Mono fase” con immagine 2018.jpg	31
4.3	Benchmark algoritmi ”Multi fase” con immagine 2018.jpg	32
4.4	Benchmark algoritmi ”Zero-Crossing” con immagine 2018.jpg	33
4.5	Immagine originale - scala di grigi - GroundTruth 3063.jpg	34
4.6	Benchmark algoritmi ”Mono fase” con immagine 3063.jpg	34
4.7	Benchmark algoritmi ”Multi fase” con immagine 3063.jpg	35
4.8	Benchmark algoritmi ”Zero-Crossing” con immagine 3063.jpg	35
4.9	Immagine originale - scala di grigi - GroundTruth 5096.jpg	36
4.10	Benchmark algoritmi ”Mono fase” con immagine 5096.jpg	36
4.11	Benchmark algoritmi ”Multi fase” con immagine 5096.jpg	37
4.12	Benchmark algoritmi ”Zero-Crossing” con immagine 5096.jpg	37
4.13	Immagine originale - scala di grigi - GroundTruth 6046.jpg	38
4.14	Benchmark algoritmi ”Mono fase” con immagine 6046.jpg	39
4.15	Benchmark algoritmi ”Multi fase” con immagine 6046.jpg	40
4.16	Benchmark algoritmi ”Zero-Crossing” con immagine 6046.jpg	41
4.17	Immagine originale - scala di grigi - GroundTruth 8068.jpg	42
4.18	Benchmark algoritmi ”Mono fase” con immagine 8068.jpg	42
4.19	Benchmark algoritmi ”Multi fase” con immagine 8068.jpg	43
4.20	Benchmark algoritmi ”Zero-Crossing” con immagine 8068.jpg	43
A.1	ClassDiagram applicazione	46

Capitolo 1

Introduzione

Il riconoscimento dei contorni è utilizzato allo scopo di marcare i punti di un’immagine digitale in cui l’intensità luminosa cambia bruscamente. In altre parole i contorni sono dei punti in cui i valori dei colori dell’immagine hanno una variazione netta.

La parola *Edge* viene utilizzata per indicare sia un punto preciso per cui l’intorno contiene una forte variazione, sia una concatenazione di questi punti che materialmente vanno a costituire il contorno dell’oggetto.

Bruschi cambiamenti delle proprietà di un’immagine spesso coincidono con eventi o cambiamenti importanti del mondo fisico di cui le immagini sono la rappresentazione. Il riconoscimento dei contorni è un campo di ricerca del trattamento delle immagini e della computer vision, in particolare della branca del riconoscimento delle caratteristiche (feature extraction) di cui però costituisce soltanto un punto di partenza.

L’operazione di estrazione e riconoscimento dei contorni genera delle immagini più dettagliate rispetto a quelle originali poiché elimina (in linea teorica) tutto ciò che non è considerato ”contorno”, conservando invece le informazioni essenziali per descrivere la forma e le caratteristiche strutturali e geometriche degli oggetti rappresentati.

Come a breve sarà mostrato, esistono diversi approcci al riconoscimento dei contorni. Lo scopo di questo progetto è approfondire e analizzare nel dettaglio gli algoritmi più noti e confrontarne le prestazioni tra loro.

Capitolo 2

Algoritmi

L'obiettivo dell'estrazione dei contorni di un'immagine, come abbiamo visto, è quello di individuare i punti in cui vi è un cambiamento netto del livello di grigio. Esistono molti metodi per fare ciò, ma la maggior parte può essere raggruppata nelle seguenti categorie.

2.1 Search-based algorithm (Gradiente)

Tali tipologie di algoritmi riconoscono i contorni individuando i massimi ed i minimi della derivata prima dell'immagine. Dal momento che stiamo lavorando su immagini 2D, ci troviamo in \mathbb{R}^2 e quindi si considera il *gradiente*:

$$\nabla(f(x, y)) = \begin{bmatrix} J_x \\ J_y \end{bmatrix} = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix}$$

ed essendo questo un semplice vettore si calcolano norma:

$$\sqrt{J_x^2 + J_y^2}$$

e direzione:

$$\arctan\left(\frac{J_x}{J_y}\right)$$

La prima componente del gradiente (derivata rispetto ad x) ovviamente rappresenterà i contorni verticali, mentre la seconda (derivata rispetto ad y) rappresenterà i contorni

orizzontali.

La lunghezza del vettore (norma) corrisponde a quanto è intensa la variazione del punto analizzato ed infatti tanto più sono grandi J_x e J_y tanto più sarà grande la distanza euclidea usata per calcolare la norma.

Il gradiente identifica quindi la direzione verso cui si incontra la variazione più forte di intensità. Di conseguenza la direzione del contorno sarà data dalla normale al vettore gradiente.

2.1.1 Operatori per il calcolo del gradiente

Come visto precedentemente, l'operazione principale per l'individuazione dei contorni con tecniche "Search-based" è il calcolo del gradiente. Tale operazione risulta essere estremamente dispendiosa dal momento che richiede il calcolo della derivata. In letteratura esistono diversi operatori adatti a calcolare il gradiente di un'immagine, come di seguito illustrato.

Nelle seguenti sezioni sono mostrati degli esempi di applicazione dei vari operatori ad un'immagine in scala di grigi. L'immagine di partenza è la celebre **Lenna**:



Figura 2.1: Immagine originale usata per gli operatori

2.1.1.1 Operatore di Roberts Cross

Il più semplice e veloce dei tre. Calcola la derivata mediante una duplice convoluzione tramite le seguenti maschere di dimensioni ($2x2$) e realizzando quindi il cosiddetto calcolo "in croce".

1	0	0	1
0	-1	-1	0
Gx		Gy	

Figura 2.2: Roberts Cross Masks

La prima rappresenta la maschera per l'asse x e quindi per i contorni orizzontali, mentre la seconda rappresenta la maschera per l'asse y e quindi i contorni verticali.

Le derivate corrispondono a:

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

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

L'utilizzo di queste maschere risulta essere estremamente veloce ma, essendo molto piccole, capita spesso che del rumore venga considerato come contorno. Per questo motivo questo operatore è particolarmente adatto a immagini con rumore praticamente assente.

Le applicazioni delle due operazioni di convoluzione, come detto precedentemente, forniscono due immagini, una con i contorni orizzontali e una con i contorni verticali.

Un esempio sono le seguenti immagini:



Figura 2.3: Convoluzione tramite maschera orizzontale di Roberts Cross



Figura 2.4: Convoluzione tramite maschera verticale di Roberts Cross

Il gradiente risultante al termine dell'applicazione dell'operatore di Roberts Cross è un'immagine del tipo:



Figura 2.5: Gradiente ottenuto dall'applicazione dell'operatore di Roberts Cross

2.1.1.2 Operatore di Sobel

Il più diffuso ed efficiente tra tutti i maggiori operatori. Utilizza maschere più grandi di dimensioni (3×3) che vengono applicate tramite duplice convoluzione, lungo l'asse x e lungo l'asse y.

-1	0	1	
-2	0	2	
-1	0	1	
Gx			
1	2	1	
0	0	0	
-1	-2	-1	
Gy			

Figura 2.6: Sobel Masks

In questo caso le applicazioni delle due operazioni di convoluzione forniscono i seguenti i risultati:



Figura 2.7: Convoluzione tramite maschera orizzontale di Sobel



Figura 2.8: Convoluzione tramite maschera verticale di Sobel

Il gradiente risultante al termine dell'applicazione dell'operatore di Sobel è un'immagine del tipo:

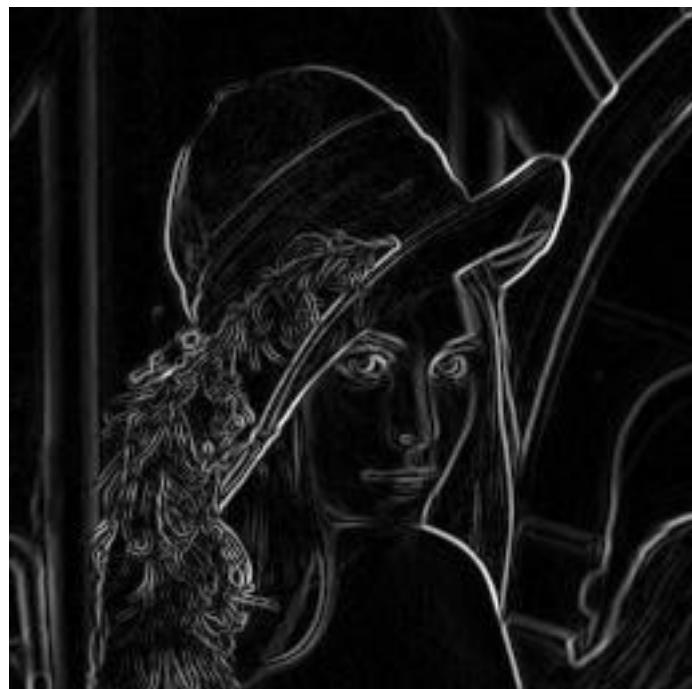


Figura 2.9: Gradiente ottenuto dall'applicazione dell'operatore di Sobel

2.1.1.3 Operatore di Prewitt

Un operatore meno diffuso ma estremamente efficiente quando si tratta di immagini prive di rumore e fortemente contrastate. I risultati ottenuti in generale sono assimilabili a quelli dell'operatore RobertCross. Diversamente da Sobel invece, questo operatore non pone in risalto i pixels che sono più vicini al centro della maschera.

Analogamente a Sobel utilizza maschere di dimensioni (3×3) che vengono applicate tramite duplice convoluzione, lungo l'asse x e lungo l'asse y.

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

G_x

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

G_y

Figura 2.10: Prewitt Masks

Un esempio dei risultati ottenuti dalle due convoluzioni è il seguente:



Figura 2.11: Convoluzione tramite maschera orizzontale di Prewitt



Figura 2.12: Convoluzione tramite maschera verticale di Prewitt

Mentre il gradiente risultante per il precedente esempio è il seguente:

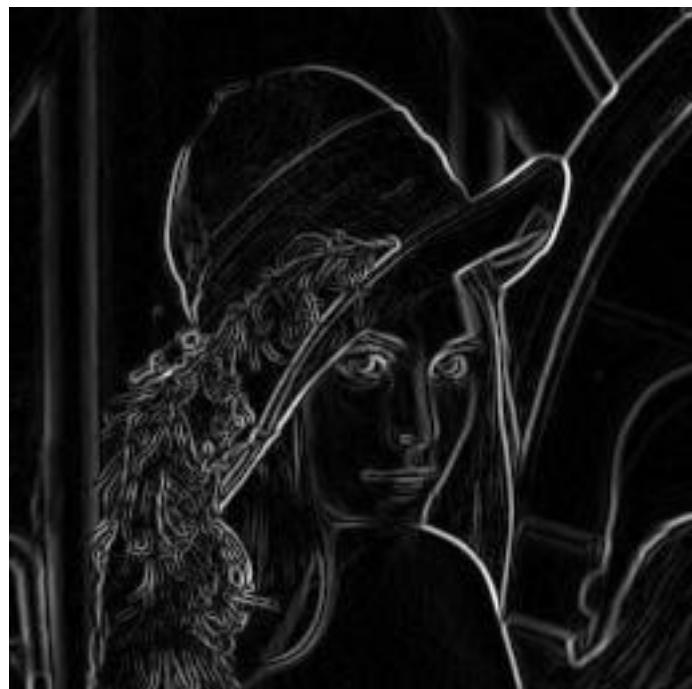


Figura 2.13: Gradiente ottenuto dall'applicazione dell'operatore di Prewitt

2.1.2 Algoritmo mono fase

Gli algoritmi che utilizzano il calcolo del gradiente in genere vedono una fase preliminare in cui viene applicato un filtro di smoothing all'immagine. Tale operazione ha un duplice scopo, quello di attenuare i cambi di intensità meno netti che individuano dettagli non corrispondenti al contorno cercato, e quello di ridurre il rumore. Senza l'utilizzo di tale fase preliminare, è chiaro che verrebbero identificati molti falsi positivi. Ci sono diversi tipi di filtri di smoothing che vengono utilizzati in queste circostanze. Uno dei più famosi, ed utilizzato nel progetto in esame, è il *filtro gaussiano*.

2.1.2.1 Gaussian Blur

L'applicazione del filtro gaussiano prevede che ogni pixel venga rimpiazzato dalla media pesata dei pixels in un suo intorno, secondo una funzione Gaussiana:

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

La dimensione del filtro e la deviazione standard in genere devono essere dipendenti l'una dall'altra in modo da garantire una copertura del 95% o superiore della probabilità. In genere si prende una dimensione del filtro che sia circa 3/5 volte la deviazione standard per ogni direzione. Andare oltre le 5 volte è dimostrato che non porta giova-menti sostanziali.

Anche in questo caso l'applicazione del filtro all'immagine avviene tramite convolu-zione. Un esempio dell'applicazione di un filtro di smoothing gaussiano con $\sigma = 3$ e conseguente dimensione pari a $5\sigma = 15$ è il seguente:

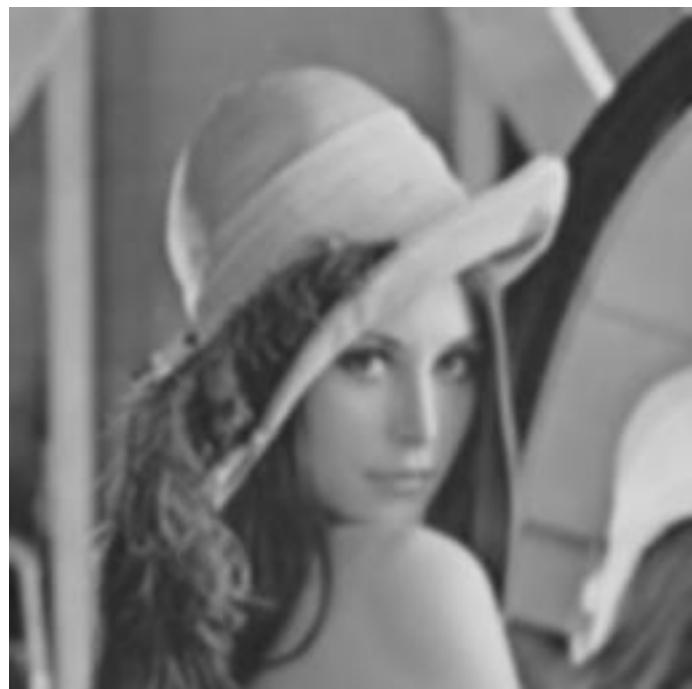


Figura 2.14: Applicazione di un filtro gaussiano con $\sigma = 3$ e dimensione = 15

2.1.2.2 Gradiente

Ottenuta l'immagine "sfuocata" si procede quindi al calcolo del gradiente mediante uno degli operatori visti in precedenza. Di seguito un confronto tra i tre operatori analizzati con e senza applicazione del blurring:

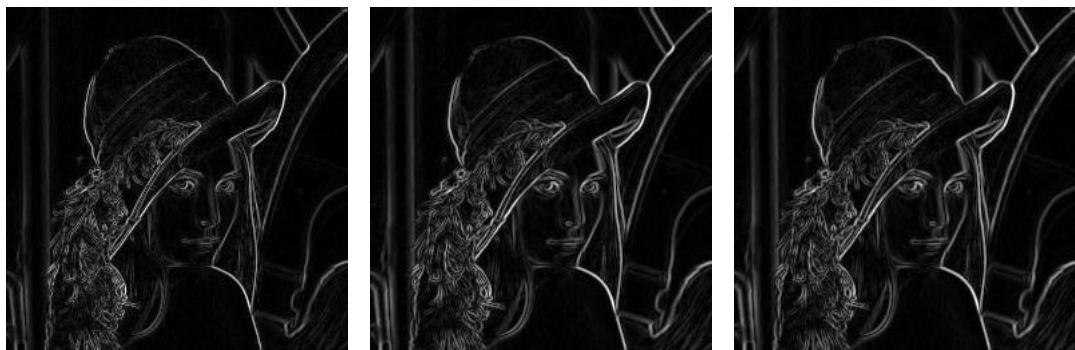


Figura 2.15: Gradiente ottenuto con Roberts Cross - Sobel - Prewitt

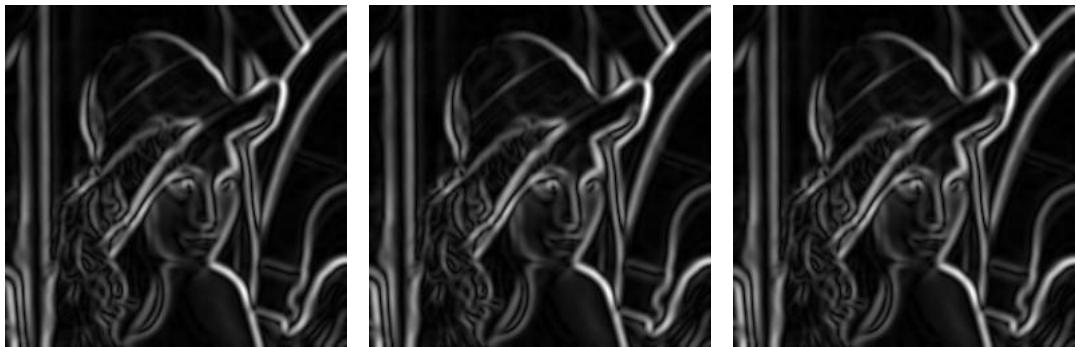


Figura 2.16: Gradiente ottenuto con Roberts Cross - Sobel - Prewitt (Blurring)

Come è possibile notare, gli operatori Sobel e Prewitt tendono ad esaltare ed illuminare di più i contorni.

Differenze più marcate sarà possibile notarle nei prossimi paragrafi quando si analizzerà l'algoritmo di Canny.

2.1.2.3 Treshold

A questo punto si stabilisce un *threshold* (soglia) e tramite questo si assegnano i pixel al contorno. A seconda della soglia selezionata si riesce a ottenere o meno la rimozione di falsi positivi relativi al rumore.

Questa soglia non è altro che un valore compreso tra 0 e 255 che rappresenta l'intensità minima (in scala di grigi) del pixel (e quindi la norma del gradiente in quel punto) che questo deve avere per far parte del contorno.

Una volta identificati i pixels tramite questa operazione, il loro valore viene normalizzato al valore della soglia.

Di seguito un confronto tra i tre operatori analizzati con e senza applicazione del blurring:



Figura 2.17: Applicazione del threshold ($= 80$) ai gradienti di Roberts Cross - Sobel - Prewitt



Figura 2.18: Applicazione del threshold ($= 80$) ai gradienti di Roberts Cross - Sobel - Prewitt (Blurring)

Come si può notare da questi esempi, gli algoritmi mono fase hanno il problema di non restituire contorni interi. Tale problema è direttamente derivabile dal fatto che i pixel restituiti sono gli unici che superano una determinata soglia e, ovviamente, non è detto che tutti i pixel di un bordo abbiano la stessa intensità.

Inoltre si può notare come, utilizzando la tecnica del blurring per rimuovere il rumore, i contorni risultino più spessi. Questo ovviamente va in contrasto con la definizione di "edge" data precedentemente, dal momento che ci si aspetterebbe lo spessore di un singolo pixel nel caso ideale.

Tutte queste problematiche vengono mitigate dagli algoritmi multi fase, come l'algoritmo di Canny mostrato nella sezione successiva.

Di seguito un esempio con tutti gli step eseguiti sia su immagine originale, sia su immagine alla quale è stato applicato un filtro gaussiano:

Sobel



Figura 2.19: Step eseguiti da un algoritmo "Mono fase" con operatore di Sobel

2.1.3 Algoritmo multi fase (Canny)

Come si è visto nelle sezioni precedenti, gli approcci mono fase sono semplici da realizzare e rapidi nel calcolo, ma non rispettano i seguenti requisiti che un edge detector dovrebbe sempre garantire.

1. **Good detection:** Minimizzare la probabilità di ottenere falsi positivi e minimizzare quella di ottenere falsi negativi;
2. **Good localization:** Il punto estratto come appartenente al contorno deve essere il più vicino possibile al punto considerato nell’immagine;
3. **Single response constraint:** Il punto restituito deve essere unico per ogni pixel riconosciuto come appartenente ad un contorno.

Come si è visto nelle precedenti sezioni, i contorni ottenuti dopo l’applicazione del gradiente risultano essere più spessi del dovuto e questo va contro il requisito della *single response constraint*.

Inoltre andando ad applicare un filtro di blurring, abbiamo visto che riduciamo la probabilità di falsi positivi migliorando la detection, ma allo stesso tempo peggiora la localizzazione e in più inspessisce i contorni. Per tale ragione bisogna trovare il giusto tradeoff per entrambi i requisiti.

L’algoritmo di Canny tenta di rispettare tutti i suddetti requisiti con un susseguirsi di fasi atte a migliorare il risultato finale.

2.1.3.1 Edge Enhancement

In questa prima fase l’algoritmo tenta di preparare l’immagine per l’applicazione delle tecniche di miglioramento successive.

Si compone di due sotto fasi, derivate dall’approccio mono fase.

Smoothing Esattamente come per gli algoritmi mono fase, anche l’algoritmo di Canny applica come primo passaggio un filtro di smoothing per ridurre il rumore prima di calcolare il gradiente.

Gradiente L’algoritmo di Canny è, come quelli mono fase visti precedentemente, basato sul calcolo del gradiente. In teoria si può usare qualsiasi maschera di quelle

mostrate in precedenza per calcolare le derivate e quindi i due vettori gradienti. Nella versione classica Canny utilizza l'operatore di Sobel per il calcolo del gradiente ma in questo progetto saranno messi a confronto i risultati ottenuti con l'utilizzo di tutti e 3 gli operatori illustrati.

Calcolato il gradiente, se ne calcola la norma e la direzione con le formule illustrate precedentemente.

2.1.3.2 Non-Maximum Suppression

Idealmente, l'immagine finale si è detto che dovrebbe avere i contorni quanto più sottili possibili e si è visto anche che, con l'applicazione del gradiente, i contorni risultano decisamente più spessi di quanto richiesto.

L'idea alla base di questa procedura è che, tra i vari punti appartenenti al contorno, si sceglie quello con intensità maggiore lungo la direzione del contorno.

Se si considerano i pixel che costituiscono l'intorno del punto analizzato, si individuano 4 direzioni, una per ogni pixel dell'intorno. Le direzioni quindi saranno identificate dai seguenti angoli: $[0^\circ, 45^\circ, 90^\circ, 135^\circ]$.

A questo punto per ogni punto identificato, si individua la direzione più vicina a quella del gradiente. Quindi si seleziona il punto che massimizza la norma del gradiente tra il punto analizzato e i due punti appartenenti alla direzione individuata. Il punto individuato è quello che massimizza la variazione di intensità lungo la direzione scelta.

Così facendo, seleziono sempre un unico punto del contorno andando a rispettare il requisito *single response constraint*.

Di seguito un esempio che mostra l'applicazione di questa tecnica ai gradienti ottenuti tramite i vari operatori.



Figura 2.20: Non-Maximum Suppression Roberts Cross - Sobel - Prewitt

Come può essere facilmente notato, e come garantito anche in letteratura, il miglior risultato lo si ottiene applicando tale procedura al gradiente ottenuto tramite l'operatore di Sobel. I contorni infatti risultano essere più definiti, luminosi e "interi" e questo miglioramento si manterrà tale anche nelle fasi successive.

2.1.3.3 Hysteresis Thresholding

Nella fase precedente abbiamo ottenuto un'immagine in cui i contorni sono molto sottili ma vi sono ancora dei pixel che non appartengono al contorno reale e che quindi potrebbero corrispondere a del rumore o a dettagli che non partecipano al contorno. A questo punto, in questa fase, bisogna effettuare una cernita di tali pixel e identificare quelli realmente appartenenti al contorno cercato. A differenza degli approcci precedenti, Canny introduce una diversa tipologia di threshold chiamata Hysteresis Thresholding che si compone di due fasi consecutive.

Double Thresholding Lo scopo di questa prima fase è quello di identificare 3 tipologie di pixel definiti come: *strong*, *weak*, *non-relevant*. L'idea è che i pixel "strong" sono quei pixel che hanno un'intensità sufficientemente elevata da garantire la partecipazione al contorno. I pixel "weak" invece sono quei pixel la cui intensità non è sufficientemente elevata per essere "strong" ma non è nemmeno così bassa da essere considerati non rilevanti.

Per compiere tale operazione si utilizzano due soglie, una alta (`highThreshold`) e una bassa (`lowThreshold`). La soglia alta serve ad identificare quei pixel con un'intensità maggiore di questa soglia e che saranno classificati come "strong". Un valore basso coincide con una maggior presenza di contorni spuri.

La soglia bassa invece serve ad identificare quei pixel la cui l'intensità è minore di questa

soglia e che saranno classificati come "non-relevant". Un valore elevato coincide con una maggiore continuità dei bordi.

I pixel compresi tra i due threshold saranno classificati come "weak" e verranno manipolati dallo step successivo (Hysteresis).

Il problema di questa metodologia è che bisognerebbe settare a mano le due soglie, e queste ovviamente dipendono dall'immagine in esame.

La tecnica per eccellenza per individuare in maniera automatica i thresholds è la "Otsu's method" che però risulta essere molto dispendiosa.

Per tale motivo in questo progetto è stata proposta una versione adattativa di tale thresholding che però risulta essere praticamente immediata rispetto al suddetto metodo. Vengono definiti due valori, "*highThresholdRatio*" e "*lowThresholdRatio*", appartenenti a $[0, 1]$ che corrispondono a dei moltiplicatori che permettono di adattare le soglie all'immagine in esame. In particolare, la soglia alta è calcolata come una determinata percentuale (*highThresholdRatio*) della massima intensità di pixel dell'immagine, mentre la soglia bassa è calcolata come una percentuale (*lowThresholdRatio*) della soglia alta.

In letteratura spesso questo approccio è proposto come alternativa all'"Otsu's Method". Le equazioni utilizzate quindi sono:

$$\text{highThreshold} = \text{highThresholdRatio} * \text{img_max}$$

$$\text{lowThreshold} = \text{lowThresholdRatio} * \text{highThresholdRatio}$$

dove *img_max* è la massima intensità dei pixel nell'immagine.

Un esempio dell'applicazione di questo metodo è il seguente:



Figura 2.21: Double Thresholding Roberts Cross - Sobel - Prewitt

Qui si può notare come tutti i pixel che nella fase precedente avevano un'intensità più bassa di altri ora sono stati considerati come "weak" e corrispondono a quelli di colore grigio scuro. Mentre tutti i pixel più luminosi nella fase precedente ora sono considerati come "strong" e sono bianchi (intensità massima). Molti altri pixel, invece, sono ora scomparsi del tutto perché classificati come "non-relevant".

Si può notare, come detto in precedenza, che l'utilizzo del filtro di Sobel, ancora una volta, contribuisce al risultato migliore.

Edge tracking by hysteresis A questo punto entra in gioco l'hysteresis vera e propria che permetta di stabilire se e quali pixel marcati come "weak" devono essere inseriti nel contorno finale oppure no. L'idea è che un contorno dovrebbe essere costituito da una linea unica e non spezzettata come quella ottenuta allo step precedente. In questo step si cerca proprio di completare le linee "promuovendo" determinati pixel classificati come "weak".

Nello specifico, la trasformazione avviene quando, nell'intorno del pixel "weak" analizzato, vi è almeno un pixel "strong", come mostrato nella figura di seguito.

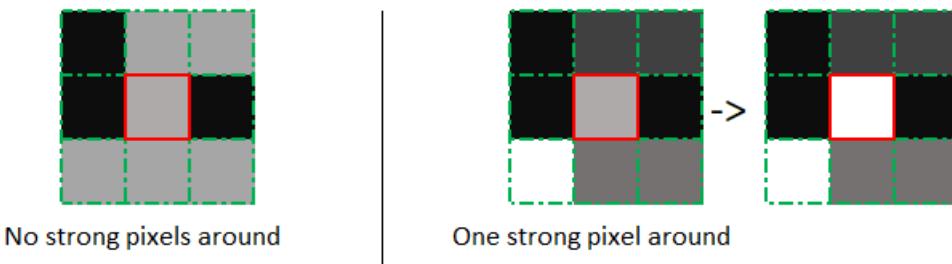


Figura 2.22: Meccanismo Hysteresis

Di seguito un esempio di applicazione di tale tecnica sulle immagini fornite allo step precedente.



Figura 2.23: Edge tracking by hysteresis Roberts Cross - Sobel - Prewitt

Come si può notare, alcuni pixel "weak" dello step precedente sono diventati "strong" mentre molti altri sono diventati "non-relevant" e quindi spenti del tutto.

Ancora una volta l'operatore di Sobel fornisce i migliori risultati.

Di seguito un esempio con tutti gli step eseguiti sia su immagine originale, sia su immagine alla quale è stato applicato un filtro gaussiano:

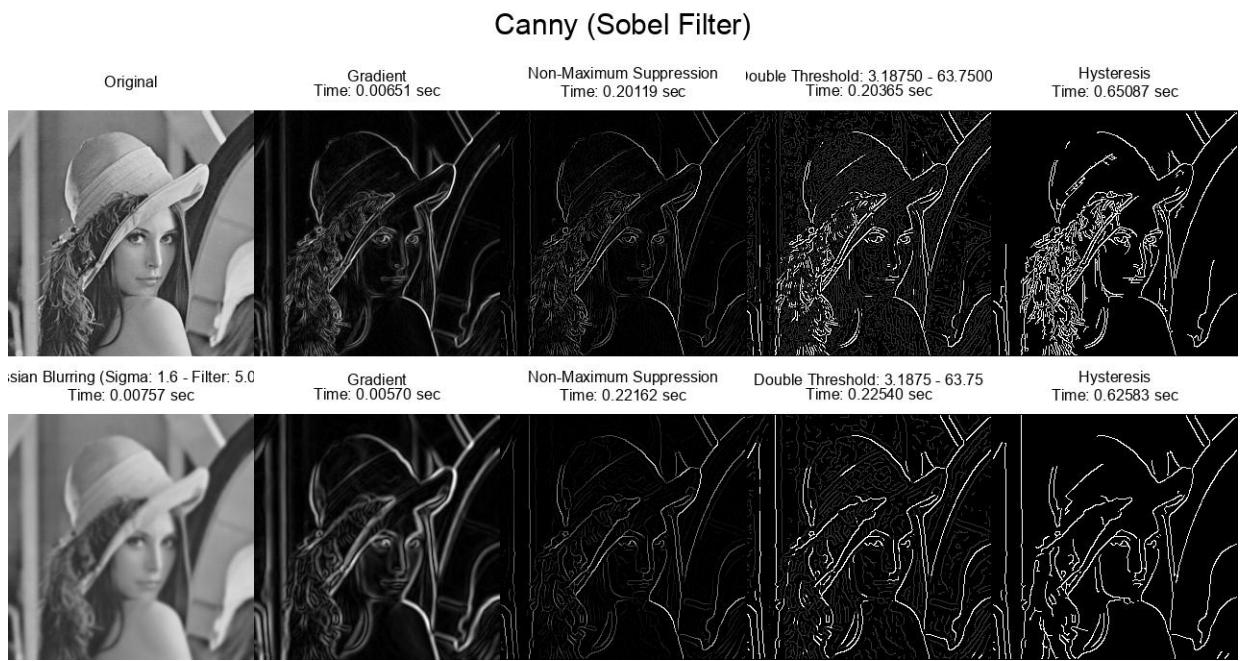


Figura 2.24: Step eseguiti da un algoritmo "Multi fase" (Canny) con operatore di Sobel

2.2 Zero-crossing algorithm (Derivata seconda)

Altri operatori per il riconoscimento dei contorni si basano sul calcolo della derivata seconda dell'intensità, che intuitivamente corrisponde a quanto cambia velocemente il gradiente da un pixel all'altro.

Nel caso in cui l'intensità varia in modo continuo, la derivata seconda si annulla nei punti di massimo del gradiente. Quindi, identificando questi punti in cui la derivata passa per lo zero, nel caso ideale, individuiamo anche i punti di massimo locale del gradiente.

Il vantaggio di questa tipologia di algoritmi è che l'operatore per il calcolo della derivata seconda è simmetrico e questo permette di trovare in una sola volta i contorni in tutte le direzioni, a differenza degli operatori precedenti basati sulla derivata prima che è direzionale.

Inoltre la ricerca degli zeri della derivata seconda è molto più semplice e veloce rispetto alle operazioni necessarie per estrarre i contorni tramite la derivata prima. In più i passaggi per lo zero di un segnale chiudono sempre i contorni fornendo quindi risultati migliori.

Il problema è che questi metodi purtroppo sono estremamente sensibili al rumore, dal momento che anche un semplice pixel può far scendere la derivata seconda a 0 e quindi essere individuato.

Un algoritmo basato su questo approccio è quello presentato nella seguente sezione.

2.2.1 Marr–Hildreth

Questo algoritmo, come detto precedentemente, richiede il calcolo della derivata seconda. Questo calcolo può essere effettuato tramite convoluzione di una maschera *Laplaciana* come, ad esempio, una di queste mostrate di seguito.

$$\begin{array}{|c|c|c|} \hline 0 & -1 & 0 \\ \hline -1 & 4 & -1 \\ \hline 0 & -1 & 0 \\ \hline \end{array} \quad \begin{array}{|c|c|c|} \hline -1 & -1 & -1 \\ \hline -1 & 8 & -1 \\ \hline -1 & -1 & -1 \\ \hline \end{array}$$

Figura 2.25: Maschere laplaciane

L'algoritmo, essendo estremamente sensibile al rumore, richiede l'applicazione di un filtro gaussiano di sharpening prima dell'applicazione del filtro Laplaciano per il calcolo della derivata seconda.

Tale necessità richiederebbe due convoluzioni consecutive sull'immagine. In realtà ciò non è vero perché, essendo la convoluzione una operazione associativa, è possibile prima applicare il filtro Laplaciano al filtro Gaussiano e soltanto successivamente applicare il filtro ottenuto all'immagine.

Quello che si ottiene con la prima convoluzione è un "*Laplacian of Gaussian*" filter.

2.2.1.1 Laplacian of Gaussian (LoG)

Questa tecnica consente di pre-calcolare la maschera da applicare all'immagine e quindi utilizzare un'unica convoluzione.

La "*LoG*" può anche essere calcolata direttamente in maniera approssimata tramite la seguente formula:

$$LoG_{\sigma}(x, y) = -\frac{1}{\pi\sigma^4} \left(1 - \frac{x^2 + y^2}{2\sigma^2}\right) e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Questo permette di ottenere il filtro in maniera ancora più veloce.

Un esempio dell'applicazione del filtro "*LoG*" è il seguente:



Figura 2.26: Applicazione filtro "*Laplacian of Gaussian (LoG)*" ($\sigma = 1.6$, $dim = 5x5$)

A questo punto non rimane che individuare quei pixel per cui all'interno dell'intorno vi è un cambio di segno della derivata, che vuol dire che la derivata seconda è passata per lo zero.

2.2.1.2 Zero-Crossing e Thresholding

In questa fase vengono analizzati tutti gli intorni dei pixel restituiti dall'applicazione del filtro "LoG" e si selezionano quei pixel per cui nell'intorno vi è un cambio di segno della derivata. Questi pixel vengono illuminati mentre tutti gli altri vengono spenti. Un esempio di applicazione di questa tecnica è il seguente:



Figura 2.27: Applicazione algoritmo di Zero-Crossing

Si può facilmente notare come vengano restituiti anche tutti quei pixel che risultavano debolmente illuminati alla fase precedente. Per tale ragione, oltre a verificare il cambio di segno della derivata seconda, si applica anche una tecnica di thresholding. A differenza dei metodi precedenti, in questo caso la soglia non viene stabilita per l'intensità dei pixel, ma per la massima differenza (in valore assoluto) tra le due derivate che portano al cambio di segno.

Così come per l'algoritmo di Canny, anche in questo caso si è scelto di utilizzare un thresholding adattivo. Si trova la media dei valori delle derivata seconde in valore assoluto, quindi se ne seleziona una determinata percentuale che costituirà il threshold.

Di seguito un esempio dell'applicazione dell'algoritmo di Zero-Crossing con thresholding:



Figura 2.28: Applicazione algoritmo di Zero-Crossing con thresholding

Dall'immagine balza subito all'occhio che le linee risultano essere continue come ci si aspettava, ma è facile anche notare come ci sia molto rumore.

Di seguito un esempio con tutti gli step eseguiti sia su immagine originale, sia su immagine alla quale è stato applicato un filtro gaussiano:

MarrHildreth (LoG)



Figura 2.29: Step eseguiti da un algoritmo "Zero-Crossing" (MarrHildreth)

Capitolo 3

Metriche

Purtroppo, in ambito di edge detection, valutare le prestazioni degli algoritmi è un lavoro molto difficile dal momento che ottenere dei contorni reali è molto difficile. A volte si tenta di effettuare una classificazione a mano, ma questo porta a risultati non sempre eccellenti dal momento che, per quanto perfetto, il contorno tracciato da un individuo potrebbe differire da quello individuabile perché magari traslato di un singolo pixel in una direzione.

Le metriche più adatte a misurare le performance degli algoritmi di edge detection sono presentate di seguito.

3.1 Mean Absolute Error (MAE)

L'errore medio assoluto, o Mean Absolute Error (MAE) è calcolato come:

$$\frac{1}{n} \sum_{j=1}^n |y - y_j|$$

Il Mean Absolute Error (MAE) è spesso conosciuto anche come L1 Loss e matematicamente rappresenta la distanza tra il valore predetto e quello effettivo. Trattandosi di una distanza, non esistono valori negativi ed infatti si utilizza il valore assoluto. Inoltre, in media, risulta essere poco sensibile agli outliers.

Nel caso delle immagini risulta molto utile perché, dato un ground truth, calcola la differenza di intensità pixel per pixel e quindi ne fa la media. Nel caso dell'edge detec-

tion, risulta essere particolarmente adatta potendo banalmente verificare se i pixel del contorno risultano illuminati o meno rispetto al ground truth.

Questa metrica offre un indice di errore che rappresenta lo scostamento dell'immagine dal ground truth, sia per posizione che per valore.

3.2 Pratt's Figure of Merit (PFOM)

È un metodo usato per fornire una comparazione quantitativa tra i vari edge detector. Il *PFOM*[2] è determinato dalla seguente espressione matematica:

$$PFOM = \frac{1}{Max(N_I, N_A)} \sum_{k=1}^{N_A} \frac{1}{1 + md^2(k)}$$

dove:

- N_I è il numero di pixel appartenenti al contorno del ground truth;
- N_A è il numero di pixel appartenenti al contorno trovato dall'algoritmo;
- m è una costante che penalizza lo spostamento del contorno individuato da quello del ground truth (generalmente pari a $\frac{1}{9}$)
- $d(k)$ è la distanza del contorno vero da quello individuato

Ovviamente il valore è sempre compreso tra 0 e 1.

3.3 Map Quality (MQ)

Il calcolo di veri positivi (TP), veri negativi (TN), falsi positivi(FP) e falsi negativi (FN) permette di tenere traccia con esattezza dei pixel individuati correttamente e di quelli invece che sono stati classificati in maniera erronea.

Sfruttando tali quantità, la metrica Map Quality[3] offre un indice di errore interessante. Gli algoritmi che individuano un piccolo numero di pixel appartenenti al contorno vengono penalizzati e tiene anche conto del numero di punti che non sono stati individuati.

La MapQuality (MQ) è calcolata nel seguente modo:

$$MQ = \frac{TP}{FP + TP + FN}$$

Capitolo 4

Analisi dei risultati

In questo progetto sono stati messi a confronto diversi algoritmi, a partire da quelli basati sul calcolo del gradiente e approccio mono fase, passando per l'algoritmo di Canny, anch'esso basato sul calcolo del gradiente ma con approccio multi fase, fino ad arrivare all'algoritmo di Marr–Hildreth basato sul calcolo della derivata seconda tramite operatore Laplaciano.

Ognuno di essi ha i suoi pro e contro, come è stato illustrato nelle sezioni precedenti. In questa sezione verranno messi a confronto i risultati ottenuti da tutti i suddetti algoritmi, applicati ad una serie di immagini di benchmark appartenenti al dataset *BSD500*[1] (Berkeley Segmentation Dataset).

Purtroppo, questi risultati non risultano essere molto utili dal momento che i contorni delle immagini ground truth sono realizzati a mano e quindi risulta essere molto difficile trovare un matching perfetto con quelli ottenuti dai vari algoritmi.

Però, quello che emerge dai risultati mostrati di seguito è che l'algoritmo di Canny (a prescindere da quale maschera si usi per il calcolo del gradiente) ottiene risultati migliori e questo è vedibile sia dai risultati forniti dalle metriche sia ad occhio nudo. In particolare, ad occhio nudo si può vedere come l'algoritmo di Canny che utilizza la maschera di Sobel per il calcolo del gradiente (ovvero l'algoritmo di Canny originale) presenta contorni più uniformi e continui rispetto agli altri due. Purtroppo questo non riesce ad essere confermato dalle metriche che valutano in maniera migliore l'utilizzo della maschera di RobertsCross. Questo è dovuto al fatto che il ground truth non rispetta fedelmente i contorni reali dell'immagine e per questo, se nell'immagine risultante sono individuati dei pixel in meno come nel caso di RobertsCross rispetto a Sobel,

CAPITOLO 4. ANALISI DEI RISULTATI

ovviamente risulteranno meno pixel classificati "erroneamente" come appartenenti al contorno.

Per quanto riguarda gli algoritmi mono fase, questi producono risultati quasi del tutto uguali se non per qualche pixel, come si può vedere dal conteggio di veri positivi, falsi positivi ecc. I risultati non sono paragonabili con quelli dell'algoritmo di Canny e nemmeno con quello di MarrHildreth perché i contorni sono estremamente spessi fornendo quindi un numero di falsi positivi estremamente elevato.

L'algoritmo di MarrHildreth, invece, risulta avere ottimi risultati paragonabili a quelli ottenuti tramite Canny nel caso in cui il rumore e gli elementi di disturbo risultano essere radi come nelle immagini "8068.jpg" e "3063.jpg". Le metriche riportano buoni risultati anche negli altri casi anche se le immagini visivamente risultano molto "rumorose". Un lato positivo è di sicuro che le linee che individua risultano essere perfettamente continue.

4.1 2018.jpg



Figura 4.1: Immagine originale - scala di grigi - GroundTruth 2018.jpg

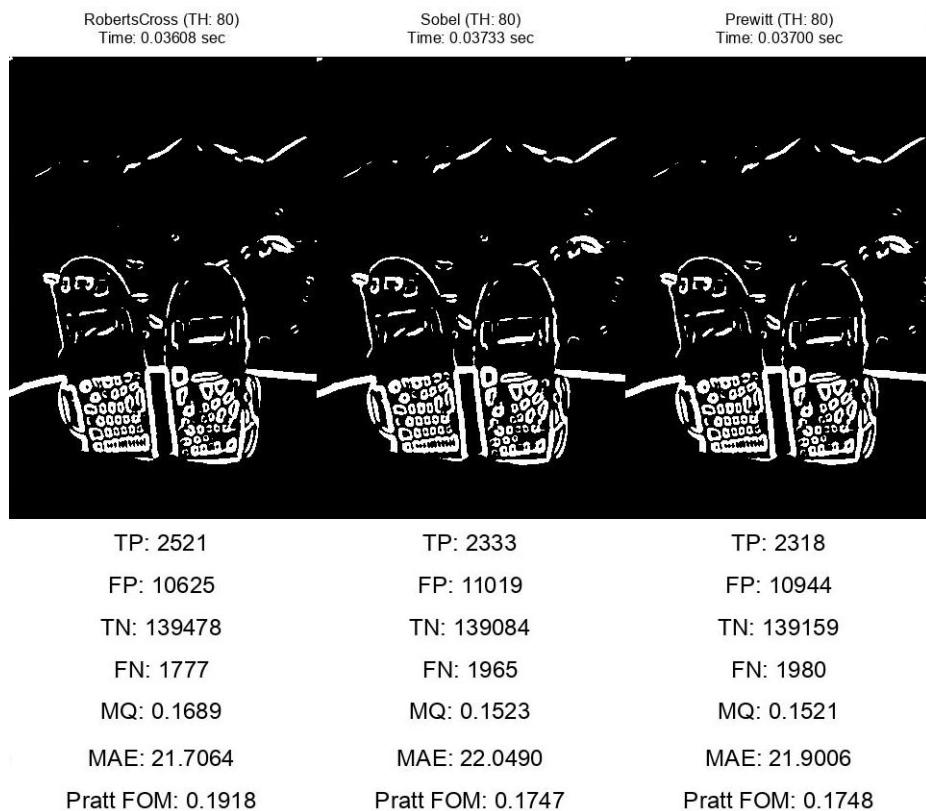


Figura 4.2: Benchmark algoritmi "Mono fase" con immagine 2018.jpg

CAPITOLO 4. ANALISI DEI RISULTATI

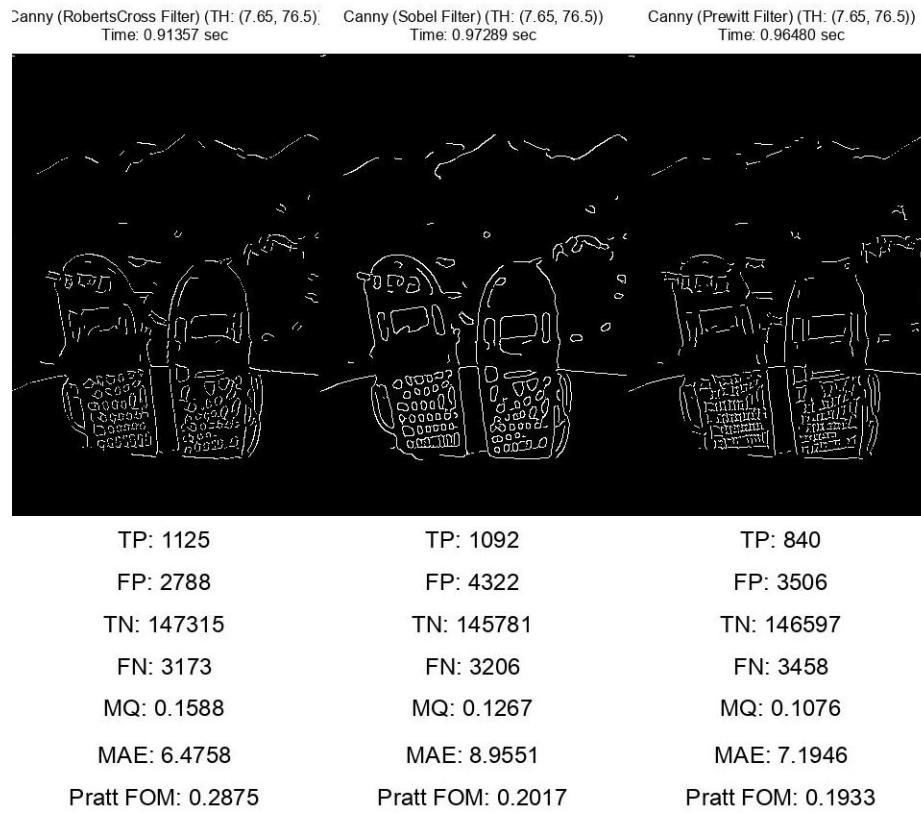


Figura 4.3: Benchmark algoritmi "Multi fase" con immagine 2018.jpg

MarrHildreth (LoG) (TH: 93.2029)
Time: 2.22736 sec



TP: 665

FP: 3352

TN: 146751

FN: 3633

MQ: 0.0869

MAE: 6.6535

Pratt FOM: 0.1655

Figura 4.4: Benchmark algoritmi "Zero-Crossing" con immagine 2018.jpg

4.2 3063.jpg



Figura 4.5: Immagine originale - scala di grigi - GroundTruth 3063.jpg

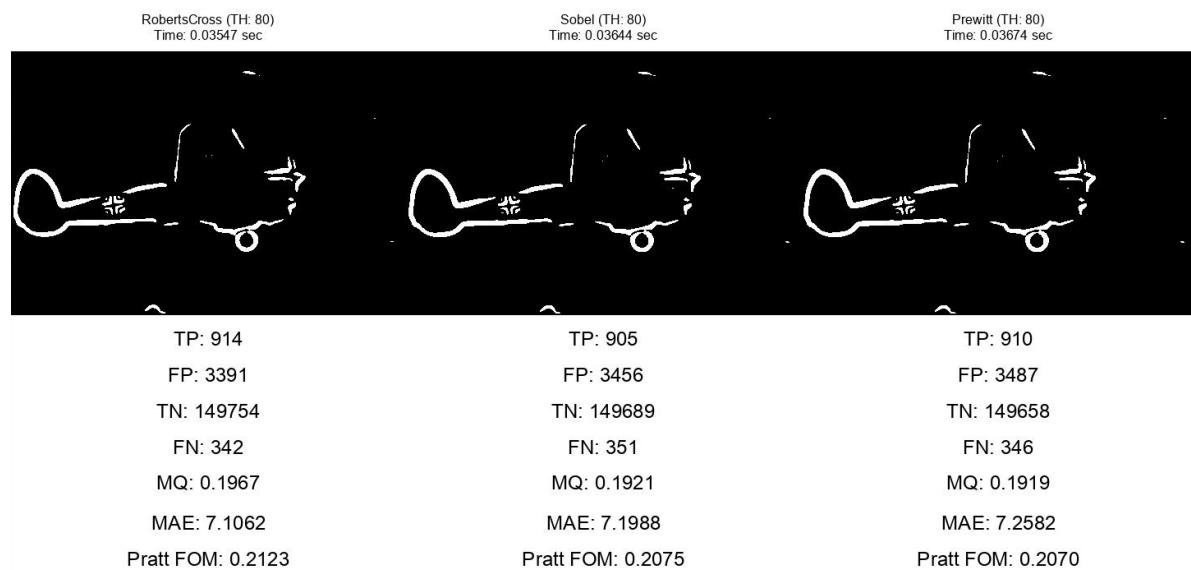


Figura 4.6: Benchmark algoritmi "Mono fase" con immagine 3063.jpg

CAPITOLO 4. ANALISI DEI RISULTATI

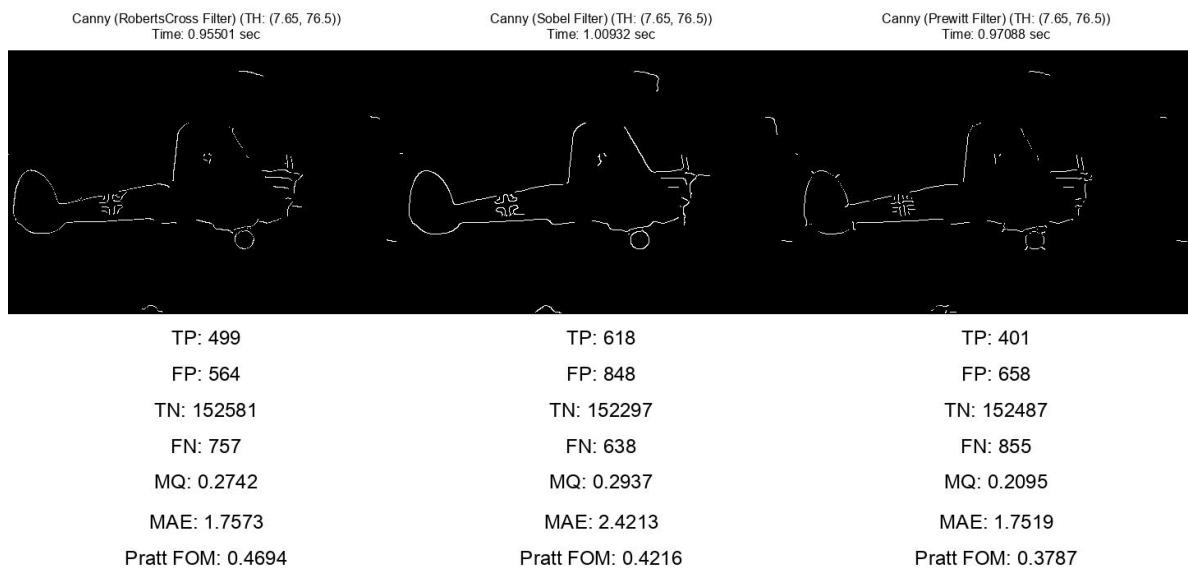


Figura 4.7: Benchmark algoritmi "Multi fase" con immagine 3063.jpg

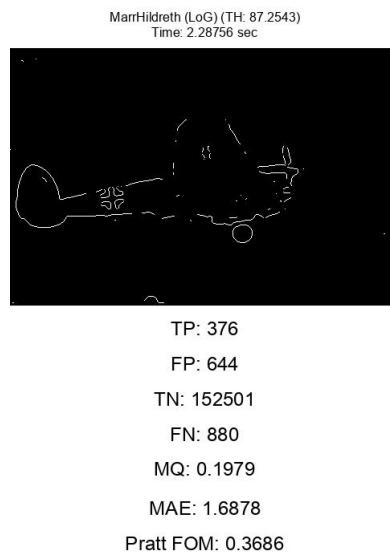


Figura 4.8: Benchmark algoritmi "Zero-Crossing" con immagine 3063.jpg

4.3 5096.jpg



Figura 4.9: Immagine originale - scala di grigi - GroundTruth 5096.jpg

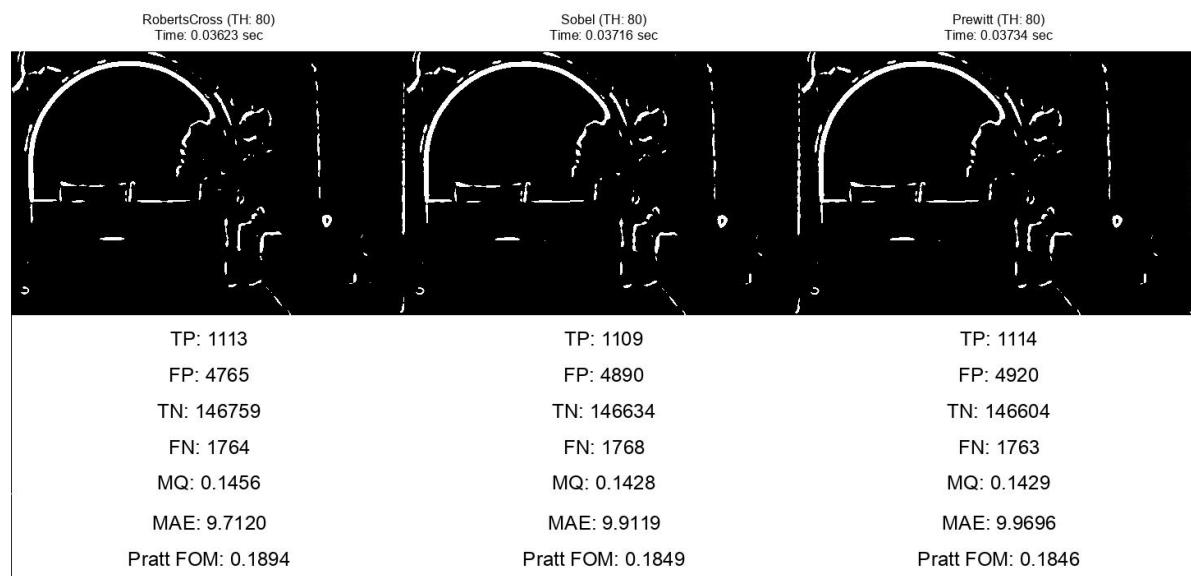


Figura 4.10: Benchmark algoritmi "Mono fase" con immagine 5096.jpg

CAPITOLO 4. ANALISI DEI RISULTATI

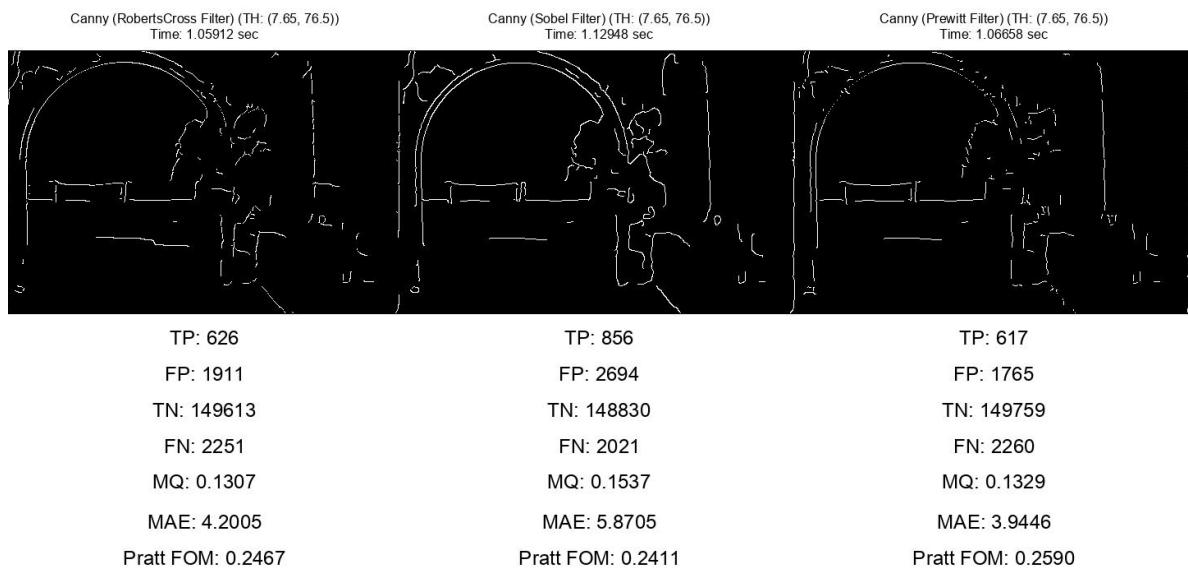


Figura 4.11: Benchmark algoritmi "Multi fase" con immagine 5096.jpg

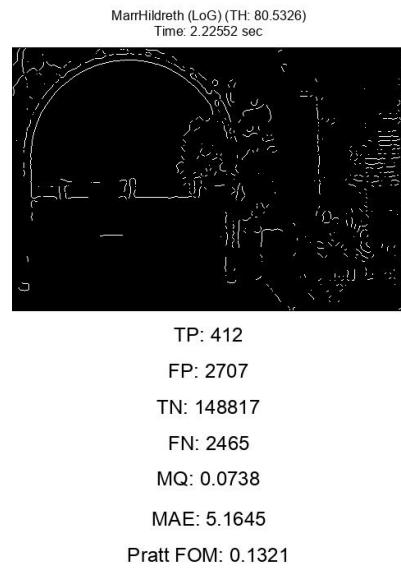


Figura 4.12: Benchmark algoritmi "Zero-Crossing" con immagine 5096.jpg

4.4 6046.jpg



Figura 4.13: Immagine originale - scala di grigi - GroundTruth 6046.jpg

CAPITOLO 4. ANALISI DEI RISULTATI

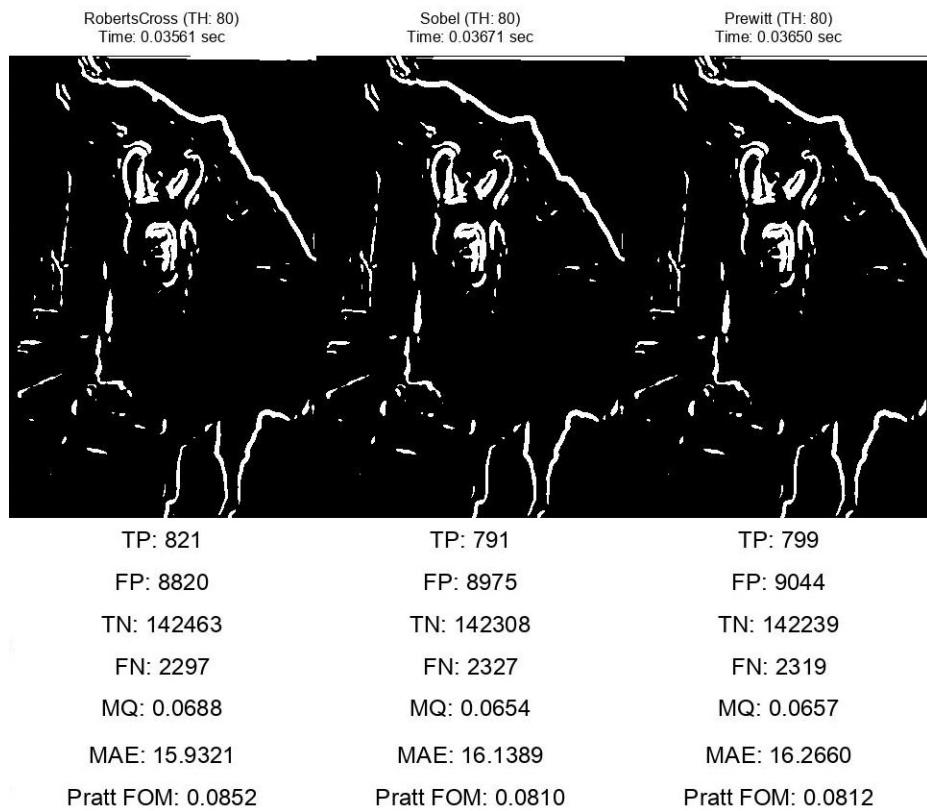


Figura 4.14: Benchmark algoritmi "Mono fase" con immagine 6046.jpg

CAPITOLO 4. ANALISI DEI RISULTATI

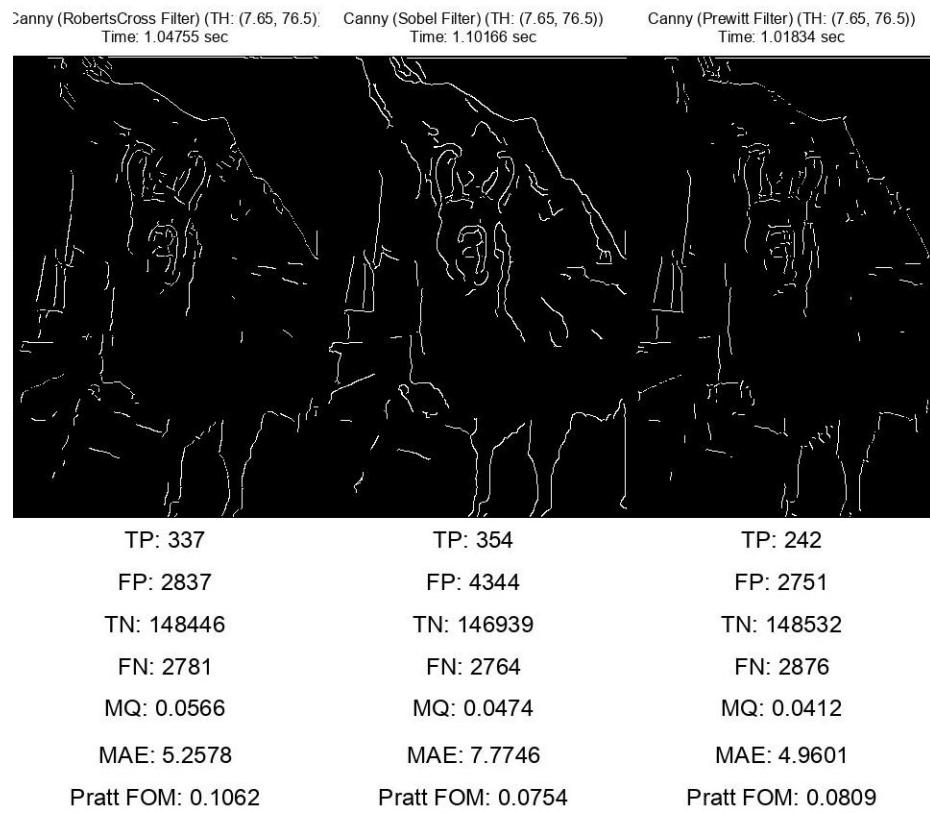


Figura 4.15: Benchmark algoritmi "Multi fase" con immagine 6046.jpg

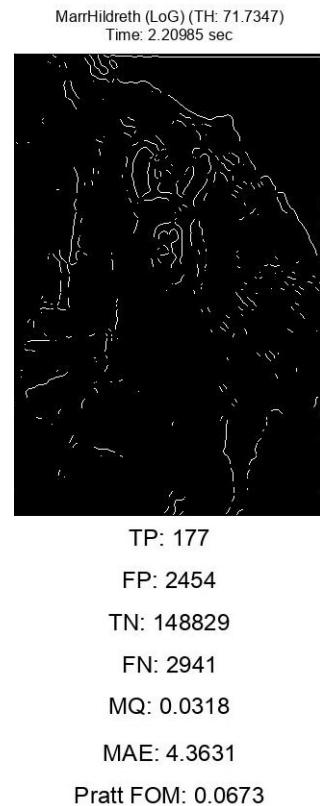


Figura 4.16: Benchmark algoritmi "Zero-Crossing" con immagine 6046.jpg

4.5 8068.jpg



Figura 4.17: Immagine originale - scala di grigi - GroundTruth 8068.jpg

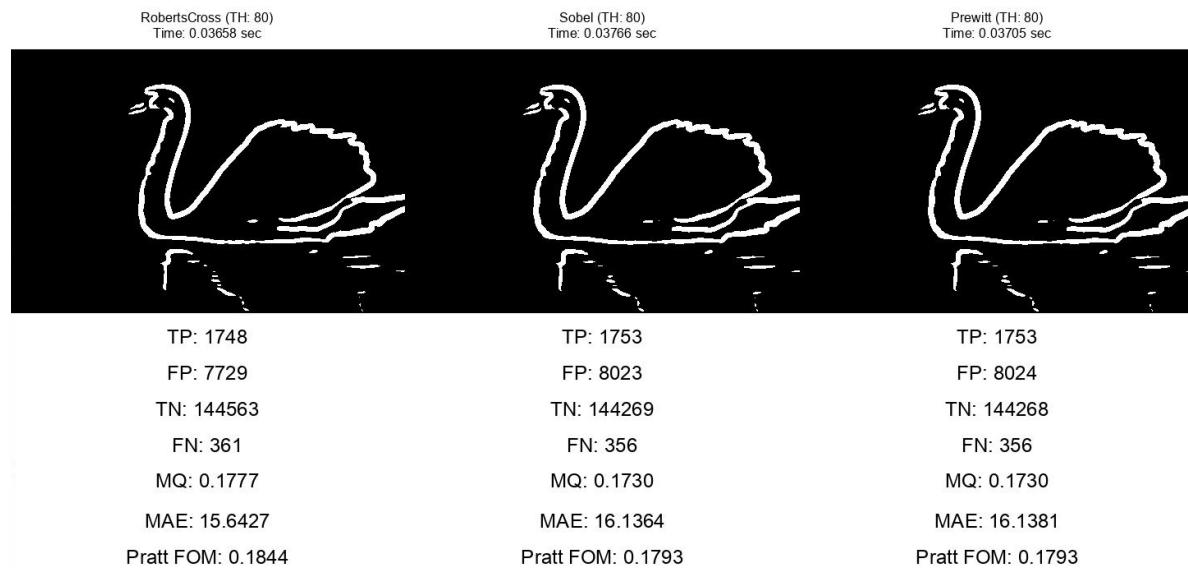


Figura 4.18: Benchmark algoritmi "Mono fase" con immagine 8068.jpg

CAPITOLO 4. ANALISI DEI RISULTATI

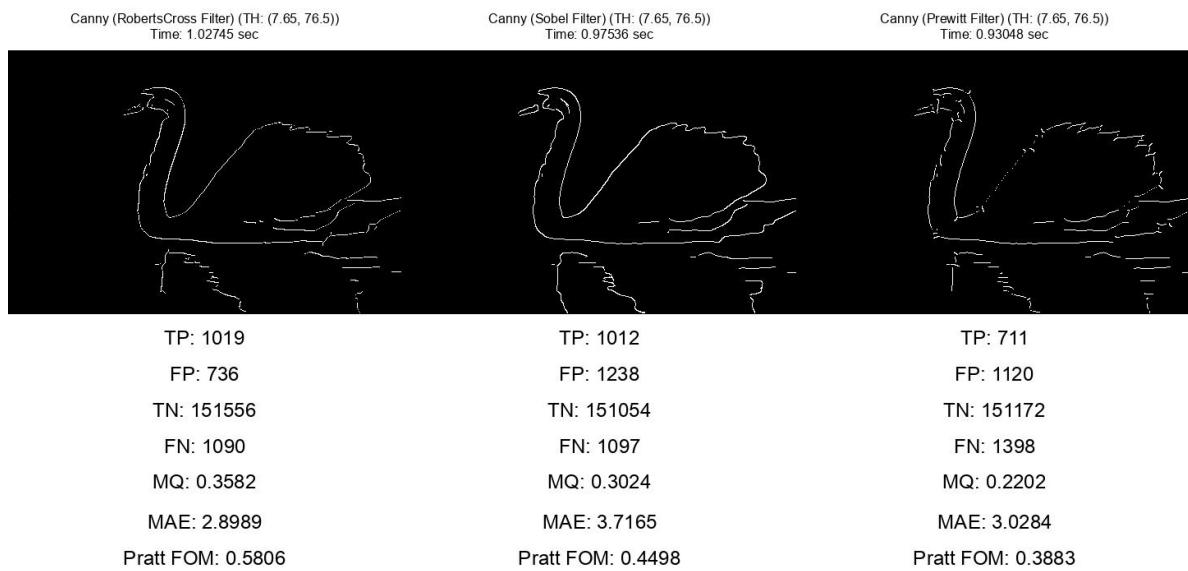


Figura 4.19: Benchmark algoritmi "Multi fase" con immagine 8068.jpg

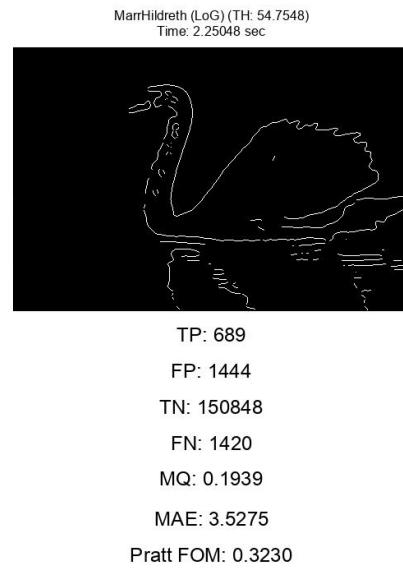


Figura 4.20: Benchmark algoritmi "Zero-Crossing" con immagine 8068.jpg

Appendices

Appendice A

Applicativo

L'applicazione è stata sviluppata in Python in ambiente virtuale ("virtualenv") al fine di rendersi indipendente dal sistema in uso. Per tale ragione, a corredo del codice sorgente viene fornito anche il file "requirements.txt" per poter ottenere tutte le dipendenze facilmente.

Il codice sorgente è disponibile anche su github al seguente link:

<https://github.com/9anSan5/EdgeDetector>

L'applicazione fornisce due principali funzionalità:

- **Benchmark** per i vari algoritmi. Tramite questa funzionalità è possibile confrontare i vari algoritmi in analisi e valutarne gli errori rispetto ad un ground truth.
- **Visualizzazione delle fasi** per ogni algoritmo. Tramite questa funzionalità è possibile visualizzare le varie fasi che si susseguono durante l'applicazione dei vari algoritmi in analisi e il confronto dei risultati ottenuti con e senza l'applicazione di un rumore gaussiano.

A.1 Design

L'applicativo è stato realizzato secondo il seguente class diagram.

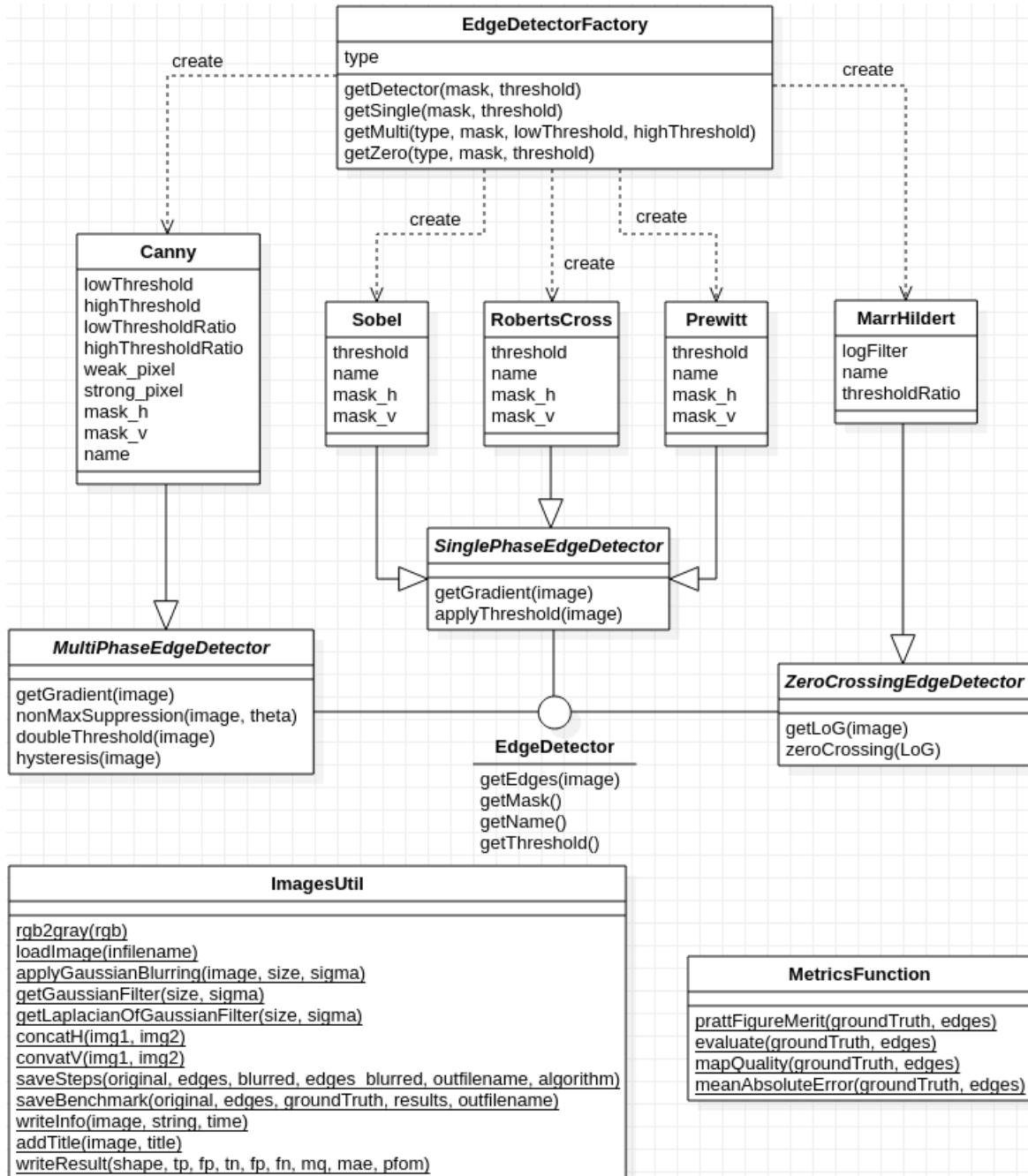


Figura A.1: ClassDiagram applicazione

È presente un'interfaccia ”*EdgeDetector*” che deve essere implementata da tutte le tipologie di detector e permette ad altri utenti di aggiungere eventualmente altri detector da testare senza dover modificare il resto del codice, considerandoli come semplici istanze di ”*EdgeDetector*”

La scelta del *Detector* da utilizzare è affidata alla classe ”*EdgeDetectorFactory*” che si occupa di istanziare in modo corretto il detector desiderato, sulla base del nome e del tipo passato in input.

Così come descritti in questo documento i vari detector sono raggruppati in ”*SinglePhaseEdgeDetector*”, ”*MultiPhaseEdgeDetector*” e ”*ZeroCrossingEdgeDetector*” che sono classi astratte. Queste vengono implementate dalle classi rappresentanti i detector specifici (”*Canny*”, ”*Sobel*”, ecc...).

La classe ”*ImageUtil*” è una classe con tutti metodi statici che fornisce funzioni auxiliarie per permettere il corretto funzionamento del tool.

La classe ”*MetricsFunction*” è anch’essa una classe con tutti i metodi statici che fornisce le funzioni per il calcolo degli errori usati dal benchmark.

A.2 Funzioni principali

In questa sezione saranno descritte le funzioni più importanti implementate, raggruppate per classi di appartenenza.

A.2.1 ImageUtil

- *getGaussian*:

INPUT: Dimensione e deviazione standard della maschera.

OUTPUT: Maschera gaussiana ottenuta come descritto precedentemente con dimensione e deviazione standard passati in input.

- *applyGaussianBlurring*:

INPUT: Immagine alla quale applicare il filtro gaussiano, dimensione e deviazione standard della maschera da generare.

OUTPUT: Immagine alla quale è stato applicato il filtro gaussiano.

- *getLaplacianOfGaussian:*

INPUT: Dimensione e deviazione standard della maschera.

OUTPUT: Filtro ”LoG”, calcolato come descritto precedentemente, con dimensione e deviazione standard passate in input.

A.2.2 SinglePhaseEdgeDetector

- *getGradient:*

INPUT: Immagine di partenza sotto forma di ”ndarray”.

OUTPUT: Magnitudine del gradiente ottenuto tramite la duplice convoluzione delle maschere orizzontale e verticale.

- *applyThreshold:*

INPUT: Magnitudine del gradiente.

OUTPUT: ndarray bidimensionale rappresentante un’immagine in bianco e nero dove gli unici pixel accesi sono quelli con magnitudine superiore al threshold.

A.2.3 MultiPhaseEdgeDetection

- *getGradient:*

INPUT: Immagine di partenza sotto forma di ”ndarray”.

OUTPUT: Magnitudine e direzione del gradiente ottenuto tramite la duplice convoluzione delle maschere orizzontale e verticale.

- *nonMaximumSuppression:*

INPUT: Magnitudine e direzione del gradiente.

OUTPUT: ndarray bidimensionale rappresentante l’immagine alla quale è stata applicata la Non-Maximum Suppression, così come è descritta in precedenza.

- *doubleThreshold:*

INPUT: ndarray bidimensionale rappresentante l’immagine su cui è stata applicata la Non-Maximum Suppression.

OUTPUT: ndarray bidimensionale rappresentante l’immagine in cui gli unici pixel visibili sono quelli ”strong” e quelli ”weak”, mentre i ”non-relevant” risultano spenti.

- *hysteresis:*

INPUT: ndarray bidimensionale rappresentante l’immagine alla quale è stato applicato il double threshold.

OUTPUT: ndarray bidimensionale rappresentante l’immagine alla quale viene applicata la tecnica dell’isteresi così come descritta precedentemente. I contorni risulteranno più ”interi” rispetto all’immagine presa in input.

A.2.4 ZeroCrossingEdgeDetection

- *getLog:*

INPUT: Immagine di partenza sotto forma di ”ndarray”.

OUTPUT: ndarray bidimensionale rappresentante l’immagine alla quale è stato applicato il filtro LoG, producendo quindi un effetto blurring e calcolando le derivate seconde.

- *ZeroCrossing:*

INPUT: ndarray bidimensionale rappresentante l’immagine alla quale è stato applicato il filtro LoG.

OUTPUT: ndarray bidimensionale rappresentante l’immagine in bianco e nero in cui gli unici pixel visibili sono quelli all’interno del relativo intorno vi è un cambio di segno della derivata (passaggio per zero) e la massima differenza tra la derivate di due pixel dell’intorno (in valore assoluto) è maggiore di una determinata soglia.

A.2.5 MetricsFunction

- *prattFigureMerit*:

INPUT: Immagine di partenza sotto forma di ”ndarray” e ground truth.

OUTPUT: Valore della metrica Pratt’s Figure of Merit calcolato come descritto precedentemente.

- *mapQuality*:

INPUT: Immagine di partenza sotto forma di ”ndarray” e ground truth.

OUTPUT: Valore della metrica Map Quality calcolato come descritto precedentemente.

- *evaluate*:

INPUT: Immagine di partenza sotto forma di ”ndarray” e ground truth.

OUTPUT: Numero di veri positivi, falsi positivi, veri negativi, falsi negativi.

- *meanAbsoluteError*:

INPUT: Immagine di partenza sotto forma di ”ndarray” e ground truth.

OUTPUT: Valore della metrica Mean Absolute Square calcolato come descritto precedentemente.

A.3 Contenuto archivio

All’interno dell’archivio si troveranno i seguenti file:

- Cartella *src* contenente i file sorgenti dell’applicativo:

Benchmark.py

Canny.py

EdgeDetector.py

EdgeDetectorFactory.py

ImageUtil.py

MarrHildreth.py

MetricsFunction.py

MultiPhaseEdgeDetector.py

Prewitt.py

RobertsCross.py

SinglePhaseEdgeDetector.py

Sobel.py

Steps.py

ZeroCrossingEdgeDetector.py

- arial.ttf: font utilizzato per la scrittura dei risultati
- ClassDiagram.png: Il class diagram di design dell'applicativo
- requirements.txt: file che contiene tutte le dipendenze per poter eseguire l'applicativo
- Cartella *Steps_Images* contenente delle immagini utili a visualizzare tutti gli step dei vari algoritmi
- Cartella *Steps_Result* contenente delle immagini che mostrano gli step eseguiti da ogni algoritmo per ogni immagine contenuta in "Steps_Images"
- Cartella *Benchmark_Images* contenente tutte le immagini da utilizzare come benchmark

Cartella *GroundTruth* contenente i ground truth per le immagini contenute in "Benchmark_Images"

- Cartella *Benchmark_Result* contenente delle immagini che mostrano le prestazioni dei vari algoritmi confrontate con il ground truth.
- Relazione.pdf corrispondente a questa relazione.
- README.md contenente la descrizione dell'applicazione e le istruzioni per l'esecuzione.

A.4 Istruzioni d'uso

Quanto riportato in questa sezione è un estratto del file README.md.

A.4.1 Prerequisiti

Per poter eseguire il codice del progetto è necessario prima di ogni altra cosa configurare l'ambiente di esecuzione.

Python 3.x Per l'utilizzo di tale applicativo è richiesta una versione di Python ≥ 3 che corrisponde alla versione utilizzata per l'utilizzo di questo framework. Non è garantita la compatibilità con versione di python inferiori, dal momento che queste sono deprecate.

Installazione su distribuzioni APT based Per poter installare Python 3.x su sistemi operativi linux basati su gestore pacchetti APT (per esempio Debian, Ubuntu, ecc.) è possibile eseguire i seguenti comandi:

```
sudo apt update  
sudo apt install software-properties-common  
sudo add-apt-repository ppa:deadsnakes/ppa  
sudo apt install python3
```

Installazione su distribuzioni RDM-YUM based Per poter installare Python 3.x su sistemi operativi linux basati su gestore pacchetti RDM/YUM (per esempio Fedora, RedHat, CentOS, ecc.) è possibile eseguire i seguenti comandi:

```
sudo yum update  
sudo yum install python3
```

pip Avendo installato Python, dovrebbe essere automaticamente presente il relativo gestore di pacchetti "pip". Qualora non fosse così, è possibile seguire le seguenti indicazioni in base al sistema operativo in uso.

Installazione su distribuzioni APT based

```
sudo apt install -y python3-pip
```

Installazione su distribuzioni RDM-YUM based

```
sudo yum install python3-pip
```

virtualenv Non è obbligatorio ma fortemente consigliato l'utilizzo di un virtual environment per l'esecuzione del codice, in maniera tale che le dipendenze siano indipendenti dal sistema.

Per l'installazione è possibile seguire i seguenti comandi:

Installazione su distribuzioni APT based

```
sudo apt install python3-venv
```

Installazione su distribuzioni RDF-YUM based

```
sudo yum install python3-virtualenv
```

A.4.2 Installazione e configurazione dell'ambiente

Una volta che tutti i suddetti prerequisiti sono soddisfatti è possibile passare alla configurazione dell'ambiente per poter eseguire correttamente l'applicativo.

Il primo step è quello di ottenere i file del progetto tramite clonazione della repository git:

```
git clone https://github.com/9anSan5/EdgeDetector.git
```

Quindi bisogna spostarsi nella cartella del progetto:

```
cd EdgeDetector
```

A questo punto inizializziamo il virtual environment e installiamo tutti i requisiti e le librerie:

```
python3 -m venv test_venv  
source ./test_venv/bin/activate  
pip3 install -r requirements.txt
```

L'ambiente virtuale è ora configurato e pronto per l'esecuzione dell'applicativo.

Per disattivare l'ambiente virtuale si può usare il seguente comando:

```
deactivate
```

A.4.3 Visualizzazione degli step per ogni algoritmo

Dopo aver clonato la repository è possibile trovare il file *Steps.py* all'interno della cartella "src". Questo file non è altro che un launcher che eseguirà una serie di operazioni atte a produrre in output delle immagini che rappresentano ognuna il susseguirsi degli step di ogni algoritmo, applicato ad ogni singola immagine di test. Viene mostrata l'applicazione degli algoritmi sia all'immagine originale che all'immagine alla quale è stato applicato in precedenza un filtro gaussiano.

Configurazione L'esecuzione del launcher può essere modificata, andando a modificare la sezione "CONFIGURE ME" del file *Steps.py* con un editor di testo.

```
##### CONFIGURE ME #####
#FilterSIGMA = 1.6
#FilterDIM = 3*FilterSIGMA

#directory = 'Steps_Images/'
#result_dir = 'Steps_Result/'

SINGLE_FASE = ["RobertsCross", "Sobel", "Prewitt"]
#MULTI_FASE = { "Canny": ["RobertsCross", "Sobel", "Prewitt"] }
#ZERO_CROSS = ["MarrHildreth"]

#single_threshold = 100
#double_threshold = [0.05, 0.25]
#zeroCrossing_threshold = 2

#####
```

Il framework è già impostato per eseguire gli algoritmi di RobertsCross, Sobel e Prewitt come detector mono fase, l'algoritmo di Canny con tutti e tre gli operatori sviluppati come detector multi fase e MarrHildreth come detector di tipo zero-crossing.

Di default vengono utilizzate le seguenti configurazioni:

Utilizza "100" come threshold per i detector mono fase, "0.05" e "0.25" come low e high threshold ratio per i detector multi fase e "0.98" come threshold per i detector di tipo zero-crossing.

Per applicare un filtro di sharpening e per generare un filtro di tipo LaplacianOfGaussian, utilizza una maschera gaussiana di dimensione (5 x 5) con deviazione standard pari a "1.6".

La cartella preimpostata che contiene le immagini da utilizzare per l'esecuzione degli algoritmi è "Steps_Images". Questa contiene alcune immagini rappresentanti attori che risultano essere particolarmente utili per mostrare i vari step degli algoritmi.

Le immagini di output verranno salvate nelle cartella "Steps_Result".

Esecuzione Per eseguire l'applicativo, dopo averlo eventualmente configurato, si può usare il seguente comando:

```
python src/Steps.py
```

A.4.4 Benchmark

Dopo aver clonato la repository è possibile trovare il file *Benchmark.py* all'interno della cartella "src". Questo file non è altro che un launcher che esegue un benchmark degli algoritmi selezionati sulle immagini scelte.

Per ogni immagine selezionata si applicano tutti gli algoritmi di edge detection configurati e quindi si confronta l'output di ognuno di questi con il ground truth relativo. I risultati vengono espressi tramite le metriche sopra descritte.

Configurazione L'esecuzione del launcher può essere modificata, andando a modificare la sezione "CONFIGURE ME" del file Steps.py con un editor di testo.

```
##### CONFIGURE ME #####
#FilterSIGMA = 2.3
#FilterDIM = 3*FilterSIGMA

#directory = 'Benchmark_Images/'
#result_dir = 'Benchmark_Result/'
#groundtruth_dir = directory+"GroundTruth/"
#SINGLE_FASE = ["RobertsCross", "Sobel", "Prewitt"]
#MULTI_FASE = { "Canny": ["RobertsCross", "Sobel", "Prewitt"] }
#ZERO_CROSS = ["MarrHildreth"]

#single_threshold = 80
#double_threshold = [0.10, 0.30]
#zeroCrossing_threshold = 2
```

Il framework è già impostato per eseguire gli algoritmi di RobertsCross, Sobel e Prewitt come detector mono fase, l'algoritmo di Canny con tutti e tre gli operatori sviluppati come detector multi fase e MarrHildreth come detector di tipo zero-crossing.

Di default vengono utilizzate le seguenti configurazioni:

Utilizza "80" come threshold per i detector mono fase, "0.10" e "0.30" come low e high threshold ratio per i detector multi fase e "0.98" come threshold per i detector di tipo zero-crossing.

Per applicare un filtro di sharpening e per generare un filtro di tipo LaplacianOfGaussian, utilizza una maschera gaussiana di dimensione (7 x 7) con deviazione standard pari a "2.3".

La cartella preimpostata che contiene le immagini da utilizzare per l'esecuzione del benchmark sui vari algoritmi è "Benchmark_Images" e al suo interno è presente la cartella contenente i relativi ground truth.

Le immagini fornite sono un piccolo sottoinsieme del dataset Barkeley Segmentation Dataset (BSD500)[1].

Le immagini di output verranno salvate nelle cartella "Benchmark_Result".

Esecuzione Per eseguire l'applicativo, dopo averlo eventualmente configurato, si può usare il seguente comando:

```
python src/Benchmark.py
```

Bibliografia

- [1] Pablo Arbelaez et al. «Contour Detection and Hierarchical Image Segmentation». In: *IEEE Trans. Pattern Anal. Mach. Intell.* 33.5 (mag. 2011), pp. 898–916. ISSN: 0162-8828. DOI: 10.1109/TPAMI.2010.161. URL: <http://dx.doi.org/10.1109/TPAMI.2010.161>.
- [2] W. K. Pratt. *Digital image processing*. Wiley, 1991.
- [3] Maher Rajab. «Performance Evaluation of Image Edge Detection Techniques». In: *International Journal of Computer Science and Security (IJCSS)* 10 (dic. 2016), pp. 170–185.