Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Informatica

Progettazione e implementazione di una applicazione didattica interattiva per il riconoscimento di oggetti basata sull'algoritmo SIFT

Relatore:
Chiar.mo Prof.
Andrea Asperti

Presentata da:
Cristiano Piemontese

Sessione II
Anno Accademico 2015/2016

Indice

Introduzione

Lo scopo di questa tesi è la creazione di un'applicazione didattica rivolta all'esposizione di un famoso algoritmo, che rientra nell'ambito della *Computer Vision*, sviluppato da David Lowe (Lowe 1999, 2004) denominato **SIFT** (*Scale Invariant Future Transform*). **SIFT** copre numerosi aspetti interessanti della disciplina e la sua strutturazione in fasi distinte si presta particolarmente bene alla spiegazione di concetti per essa fondamentali quali la nozione di *feature*, la loro individuazione, il problema di darne una descrizione che sia indipendente dal sistema di riferimento, le euristiche per confrontare efficientemente tra loro migliaia di *feature* diverse, cosi come le tecniche che permettono di ricostruire la trasformazioni topologiche che sono avvenute tra l'immagine sorgente e quella target.
L'obiettivo finale era quello di ricreare un'implementazione di **SIFT** in un linguaggio particolarmente semplice (Python) e che in particolare permettesse all'utente di interagire con l'applicazione in alcune fasi cruciali dell'algoritmo, configurando dinamicamente alcuni parametri ed enfatizzandone l'impatto sul funzionamento complessivo.

Nel **capitolo 1** verrà spiegato brevemente cos'è la *Computer Vision* e come **SIFT** si inserisce nel suo panorama, definendo inoltre alcuni termini chiave utili a comprendere meglio l'elaborato.

Nel **capitolo 2** verranno descritte in dettaglio le fasi di cui **SIFT** si compone e le idee su cui si basa.
Nel **capitolo 3** verrà presentata l'implementazione di **SIFT**.

Le definizioni che verranno date in questo elaborato, (ad esempio definizioni di funzioni) a meno che non sia specificato diversamente, sono basate sull'articolo del 2004 di David Lowe, "Distinctive image features from scale-invariant keypoints".

Capitolo 1

Computer Vision

La *Computer Vision* è una branca dell'Informatica il cui obiettivo "è quello di creare un computer che può vedere il mondo esterno e capire cosa sta succedendo."[1] (Ikeuchi, Matsushita e Kawakami 2014, p. v)

In altre parole, la disciplina si occupa di creare e discutere algoritmi in grado di acquisire, elaborare e analizzare immagini e sequenze video. La disciplina si è sviluppata a partire dagli anni settanta circa ed è arrivata a comprendere varie sotto-discipline, come ad esempio: l'*image restoration*, che si occupa di ottenere immagini pulite a partire da immagini corrotte o con un qualche tipo di rumore; il *video tracking*, che riguarda i processi per localizzare oggetti in movimento; l'*object recognition*, cioè il riconoscimento di oggetti in immagini o sequenze video.
Per capire l'importanza di queste discipline basti pensare ai recenti sforzi fatti da Google e altre multinazionali per creare dei veicoli che possano guidare in modo autonomo. È chiaro come questo tipo di tecniche sia necessario per ottenere obiettivi del genere.

[1] "[its goal] is to make a computer that can see its outer world and understand what is happening." (trad. mia)

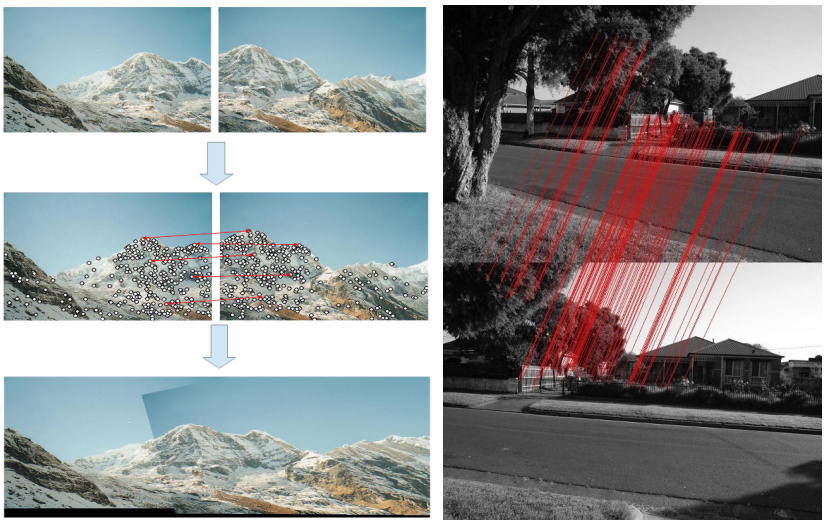

Figura 1: A sx un esempio di ricostruzione di panorama, a dx un esempio di match di feature

SIFT (*Scale Invariant Feature Transform*) è un algoritmo di *Computer Vision* sviluppato da **David G. Lowe**, che attualmente lavora come *Senior Research Scientist* per Google (Seattle)[2]. Presentato in una prima versione nel 1999 e successivamente riproposto (in versione aggiornata) nel 2004, in un articolo che è diventato uno dei più citati di tutta la *Computer Vision* (∼35,000 citazioni su Google Scholar), è diventato un algoritmo chiave fra quelli che si basano sull'individuazione di *feature* locali (per *feature* si intende un punto di interesse su di un'immagine, con certe proprietà che verranno definite successivamente). Il successo si deve a vari fattori fra cui l'affidabilità e la robustezza dei **descrittori** di *feature* da esso creati. Un **descrittore** non è altro che un vettore che tenta di descrivere un intorno di una data *feature* in un modo che sia indipendente dal sistema di riferimento e che permetta di ritrovare, se presenti, stesse *feature* in immagini differenti. Un altro fattore del successo di **SIFT** è dovuto al fatto che è stato uno fra i primi algoritmi di *feature detection* che potesse ritrovare le stesse *feature* a scale diverse, cioè

[2]Cfr. pagina ricercatore: `http://www.cs.ubc.ca/~lowe/home.html`

feature che fossero **invarianti per cambi di scala**. Algoritmi precedenti, come ad esempio *Harris Corner Detector* (Harris e Stephens 1988), non sono invarianti per cambi di scala e nonostante questo non sia un requisito necessario in tutte le applicazioni è tuttavia molto importante ad esempio se si vuole effettuare *object recognition*.

Un esempio dell'importanza di **SIFT** si può vedere dal fatto che, da quando è stato descritto, sono stati presentati molti altri algoritmi basati su idee simili e i cui **descrittori** sono definiti **SIFT-like**, come quelli di **GLOH** (Mikolajczyk e Schmid 2005) e **SURF** (Bay et al. 2008).

SIFT ha applicazioni in vari ambiti della *Computer Vision*, come la ricostruzione 3D di viste multiple e i già menzionati video tracking e object recognition (eempi di applicazioni sono visibli in figura 1).

In generale, ciò di cui lo svolgimento di questi compiti necessita è di poter descrivere gli oggetti che compaiono in un immagine o frammento video in un modo che permetta a un computer di poterli distinguere autonomamente e in modo affidabile e **SIFT** si presta bene per la loro risoluzione per via delle caratteristiche già menzionate.

Capitolo 2

SIFT

SIFT è suddivisibile in due momenti fondamentali: *detection* e *description* di *feature*. Nonostante queste due parti siano strettamente legate e solo insieme costituiscano **SIFT**, è bene sottolineare che sono anche utilizzabili separatamente e in combinazione con altri algoritmi (ad esempio è possibile creare **descrittori SIFT** a partire da bordi trovati con *Canny Edge Detector* (Canny 1986) o creare **descrittori SURF** usando le *feature* trovate con **SIFT**).

2.1 Fase di detection

La prima fase è quella di *detection*. Sebbene ogni *detector* miri a trovare uno specifico tipo di *feature* (ad esempio *Canny Edge Detector* per i bordi o *Harris Corner Detector* per gli angoli), vi sono delle caratteristiche comunemente desiderabili. In particolare, le *feature* trovate devono essere:

- **locali**, perché meno dipendenti da deformazione e occlusione dell'oggetto. Inoltre in un'immagine sono presenti in grandi quantità, cosa

che permette di distinguere meglio oggetti diversi;

- **invarianti**, ad esempio invarianti per cambi di scala, posizione e orientamento (dipende dal *detector* usato);
- **distintive**, ossia che permettono di avere un grado di riconoscimento alto, mantenendo basso il numero di falsi positivi
- **ripetibili**, dev'essere possibile ritrovare le *feature* in più immagini.

2.1.1 Detection degli estremi dello spazio-scala

Il primo passo per trovare le *feature* consiste nel creare la rappresentazione in **spazio-scala** dell'immagine da processare.
Va allora definita la **teoria dello spazio-scala**. Secondo la definizione di Lindeberg (Lindeberg 2008, p. 1) "la teoria dello spazio scala è un *framework* per la rappresentazione multi-scala di immagini"[1].
Poter fare ciò è essenziale per varie ragioni: gli oggetti del mondo reale, che sono quelli rappresentati nelle immagini che vengono elaborate, sono per loro natura multi-scala. Basti pensare a concetti come quello di albero, foresta e struttura molecolare: ognuno ha senso a una scala appropriata (Lindeberg, 2008, p. 2). Inoltre

> "[...] in assenza di ogni informazione preliminare su quali scale siano appropriate per un dato compito visivo, l'unico approccio ragionevole è rappresentare i dati su scale multiple."[2] (ibid., p. 2).

[1] "scale-space theory is a framework for multiscale image representation" (trad. mia)

[2] "[...] in the absence of any prior information about what scales are appropriate for a given visual task, the only reasonable approach is to represent the data at multiple scales." (trad. mia)

La **teoria dello spazio-scala** è una teoria matematica e per costruire la rappresentazione in **spazio-scala** di un'immagine va per prima cosa definita che cos'é matematicamente un'immagine: è una funzione $I(x, y)$ che date le coordinate (x, y) di un punto ritorna un valore associato a quel punto.
La **rappresentazione in spazio-scala** di un immagine è allora definita come la funzione $L(x, y; \sigma)$ t.c.:

$$L(x, y; \sigma) = G(x, y; \sigma) * I(x, y)$$

dove $*$ è l'operatore di convoluzione in x e y, σ è la deviazione standard, che rappresenta il parametro di scala, e G è la funzione Gaussiana così definita:

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

L'uso della funzione Gaussiana è motivato, intuitivamente, dal fatto che la sfocatura che permette di ottenere può essere vista come se venissero allontanati gli oggetti rappresentati. In modo più formale:

> "I requisiti, o assiomi, che specificano l'unicità del kernel Gaussiano sono fondamentalmente la linearità e l'invarianza per *shift* spaziale combinate con differenti modi di formalizzare la nozione che le strutture a scale più grossolane dovrebbero essere legate a strutture a scale più fini in un modo ben definito; il metodo di sfocatura non dovrebbe creare nuove strutture. (Lindeberg 1994, p. 3)."[3]

[3] "The requirements, or axioms, that specify the uniqueness of the Gaussian kernel are basically linearity and spatial shift invariance combined with different ways of formalizing the notion that structures at coarse scales should be related to structures at finer scales in a well-behaved manner; new structures should not be created by the smoothing method." (trad. mia)

Quindi la funzione Gaussiana è l'unico kernel possibile per costruire lo **spazio-scala**.

L'obiettivo principale di **SIFT** è quello di trovare *feature*, dette anche *keypoint* (letteralmente "punti chiave"), che siano stabili nello **spazio-scala**. Per fare ciò Lowe (Lowe 1999) ha proposto di usare come *keypoint* gli **estremi** della **Differenza di Gaussiane** convoluta con l'immagine, cioè la funzione $D(x, y; \sigma)$, definita nel seguente modo:

$$\begin{aligned} D(x, y; \sigma) &= (G(x, y; k) - G(x, y; \sigma)) * I(x, y) \\ &= (G(x, y; k) * I(x, y)) - (G(x, y; \sigma) * I(x, y)) \\ &= L(x, y; k) - L(x, y; \sigma). \end{aligned}$$

La scelta di questa funzione è dovuta a vari motivi: L va calcolata in ogni caso poiché verrà usata nella parte di assegnazione degli orientamenti e di *feature description*. Una volta che si ha L è possibile calcolare D con una semplice sottrazione fra immagini. Inoltre la **Differenza di Gaussiane** (abbreviata in DoG dall'inglese *Difference-of-Gaussian*) permette di approssimare bene il **Laplaciano della Gaussiana** (abbreviato in LoG dall'inglese *Laplacian-of-Gaussian*) normalizzato a scala, $\sigma^2 \nabla^2 G$. Infatti usando l'equazione del calore si può vedere che:

$$\sigma \nabla^2 G = \frac{\partial G}{\partial \sigma} \approx \frac{G(x, y; k\sigma) - G(x, y; \sigma)}{(k-1)\sigma}$$

e quindi:

$$(k-1)\sigma^2 \nabla^2 G \approx G(x, y; k\sigma) - G(x, y; \sigma)$$

con $(k-1)$ che è costante su tutte le scale e non influenza la *detection* degli **estremi**.

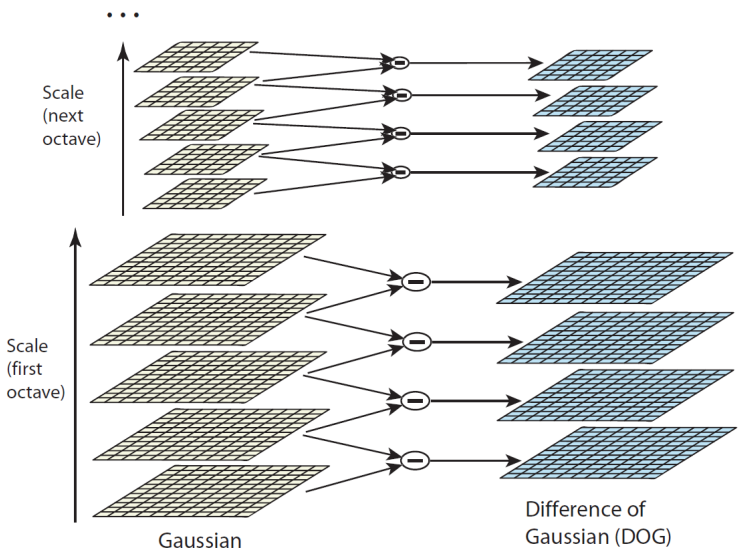

Figura 2: Esempio di piramide Gaussiana [*immagine tratta da (Lowe 2004, p. 6)*]

Un modo per costruire D efficientemente, sempre esposto da Lowe (Lowe 2004, p. 7), è quello di utilizzare il concetto di **piramide Gaussiana**, una struttura composta da immagini. Alla base della piramide si trova l'immagine iniziale, che supponiamo essere di dimensione $2^n \times 2^n$, da cui parte la costruzione. Successivamente sopra la base vengono costruiti un certo numero di livelli, chiamati **ottave**, di dimensione $2^{n-o} \times 2^{n-o}$ dove o è il numero del livello. Ogni livello successivo alla base è ottenuto sfocando (con un filtro Gaussiano, non a caso) e dimezzando l'immagine del livello precedente. In particolare la **piramide Gaussiana** usata in **SIFT** è così composta: ogni ottava o è divisa in s intervalli, detti **scale**, e per ogni ottava vengono generate $s + 3$ immagini sfocate, ognuna creata sfocando l'immagine precedente, in modo così da coprire un'intera ottava quando si andranno a cercare gli estremi. A ogni cambio di ottava la sfocatura di base σ raddoppia mentre a ogni cambio di scala la sfocatura aumenta di k volte, con $k = 2^{1/s}$, rispetto a quella dell'immagine precedente.

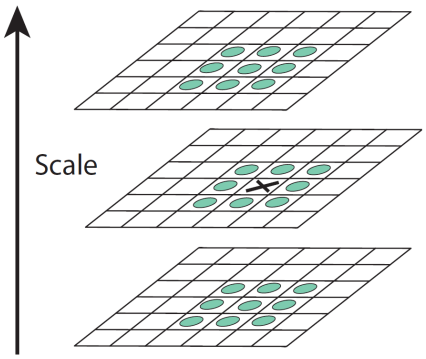

Figura 3: Ricerca degli estremi [*immagine tratta da (Lowe 2004, p .7)*]

Quindi, ad esempio, se si ha che $o = 2$, $s = 2$ (quindi $k = \sqrt{2}$) e come sfocatura iniziale si ha un certo σ_0, ogni ottava avrà 5 immagini con le seguenti sfocature: $\sigma_0, k\sigma_0, k^2\sigma_0, k^3\sigma_0, k^4\sigma_0$ per la prima ottava; $k^2\sigma_0, k^3\sigma_0, k^4\sigma_0, k^5\sigma_0, k^6\sigma_0$ per la seconda (notare che $k^2 = 2$ e quindi la prima sfocatura della seconda ottava è in effetti $2\sigma_0$).
Dopo aver costruito una **piramide Gaussiana** così fatta, ogni livello è sottratto al precedente per creare un livello della **piramide di DoG**, come mostrato in figura 2. In questo modo si otterrano $s+2$ livelli per ogni ottava della piramide.

Si può allora procedere a cercare gli **estremi** nella piramide appena ottenuta. Il metodo usato è quello di confrontare ogni punto di ogni livello (cioè di $D(x, y; \sigma)$ per opportuni σ) con gli otto punti attorno a lui nella sua scala e i nove nella scala superiore e inferiore, come mostrato in figura 3.
È facile vedere perchè è necessario creare $s + 3$ immagini sfocate per ogni ottava della **piramide Gaussiana**: dal momento che per individuare un estremo dobbiamo confrontare un punto con i punti delle immagini con scala superiore e inferiore (rimanendo nella stessa ottava), saranno necessarie al-

meno 3 immagini per ogni ottava della **piramide di DoG** per poter poterlo fare (e il valore minimo di s è 1).
Un punto chiave del processo di *detection* degli **estremi** è la scelta dei valori σ_0, o ed s, che è discussa in dettaglio da Lowe (Lowe 2004, pp. 7-10) e che è fondamentalmente sperimentale. I valori da lui scelti sono tali da permettere ottenere i risultati migliori mantenendo l'algoritmo efficiente e sono $\sigma_0 = 1.6$, $s = 3$. I numero di ottave può essere determinato in base alla dimensione dell'immagine da elaborare (e cioè $log_2(min(altezza, larghezza))$ dove $altezza$ e $larghezza$ sono le dimensioni dell'immagine) oppure essere fissato ad un certo valore scelto arbitrariamente.

2.1.2 Interpolazione degli estremi

Il secondo passo della fase di *detection* consiste nel passaggio dal discreto al continuo. Fino a ora infatti sono stati trovati **estremi** dello **spazio discreto**, cioè valori come $(x, y, \sigma) = (22, 134, 2\sigma_0)$, $(x, y, \sigma) = (42, 1729, 2^{3/2}\sigma_0)$. Tuttavia questi valori sono poco accurati: per ottenere risultati migliori si effettua quindi il *fitting* di una funzione quadratica in 3D in modo da determinare la posizione dell'estremo tramite interpolazione.
La funzione usata è l'espansione di Taylor della funzione $D(x, y; \sigma)$, spostata in modo tale che l'origine sia l'estremo da interpolare:

$$D(\mathbf{x}) = D + \frac{\partial D^T}{\partial \mathbf{x}}\mathbf{x} + \frac{1}{2}\mathbf{x}^T\frac{\partial^2 D}{\partial \mathbf{x}^2}\mathbf{x}$$

con D e le sue derivate che sono valutate nell'estremo da interpolare e $\mathbf{x} = (x, y, \sigma)^T$ che è lo scarto dall'estremo.
Prendendo la derivata di questa funzione rispetto a $\mathbf{x}$ e ponendola a 0 si

ottiene la posizione dell'estremo interpolato:

$$\hat{\mathbf{x}} = -\frac{\partial^2 D}{\partial \mathbf{x}^2}^{-1} \frac{\partial D}{\partial \mathbf{x}}$$

Per prima cosa l'Hessiana ($\frac{\partial^2 D}{\partial \mathbf{x}^2}$) e le derivate parziali vengono calcolate con il metodo delle differenze finite. Successivamente, una volta risolto il sistema lineare in tre incognite e ottenuto $\hat{\mathbf{x}}$, ci sono varie possibilità in base al valore ottenuto e cioè:

- se $\hat{\mathbf{x}}$ ha un valore superiore a 0.5 in una qualsiasi delle sue direzioni, si aggiunge all'estremo da interpolare e si effettua l'interpolazione nel nuovo punto,
- se $\hat{\mathbf{x}}$ ha valori inferiori a 0.5 in ogni direzione è stato trovato l'estremo,
- se dopo un certo numero di iterazioni non è stato trovato l'estremo, si scarta il punto.

Lowe (Lowe 2004, pp. 11, 12) effettua un ulteriore filtraggio dei *keypoint*: il primo filtraggio riguarda il contrasto e serve a scartare *keypoint* con contrasto troppo basso. Vengono infatti scartati estremi per cui $|D(\hat{\mathbf{x}})| < 0.03$ (dove 0.03 come sempre è stabilito sperimentalmente da Lowe). Il secondo filtraggio serve invece ad attutire il fatto che la funzione **DoG** dà risposte troppo forti lungo i bordi degli oggetti. Questo caratteristica porta ad ottenere *keypoint* troppo sensibili al rumore e che vanno eliminati. Lowe osserva che

> "[un] picco poco definito nella differenza di Gaussiane avrà una curvatura principale grande lungo i bordi ma piccola nella direzione perpendicolare"[4] (ibid., p. 12).

[4] "A poorly defined peak in the difference-of-Gaussian function will have a large principal curvature across the edge but a small one in the perpendicular direction." (trad. mia)

Per poter identificare un picco poco definito allora si può verificare se il **rapporto** delle curvature principali sia sotto una certa soglia o meno. Per calcolare questo rapporto si usa una matrice Hessiana $\mathbf{H}$ 2×2 calcolata, sempre usando il metodo delle differenze finite, alla locazione e scala del *keypoint*. Non è tuttavia necessario calcolare gli autovalori di $\mathbf{H}$, infatti sia α l'autovalore più grande, β quello più piccolo e $r = \frac{\alpha}{\beta}$ il loro rapporto. Si ha allora che:

$$\frac{\mathrm{Tr}(\mathbf{H})^2}{\mathrm{Det}(\mathbf{H})} = \frac{(D_{xx} + D_{yy})^2}{D_{xx}D_{yy} - D_{xy}^2} = \frac{(\alpha + \beta)^2}{\alpha\beta} = \frac{(r+1)^2}{r}$$

In questo modo il rapporto delle curvature principali è sotto una certa soglia e cioè se

$$\frac{\mathrm{Tr}(\mathbf{H})^2}{\mathrm{Det}(\mathbf{H})} < \frac{(r+1)^2}{r}$$

Il valore di r è stabilito sperimentalmente, Lowe (Lowe 2004, p. 12) propone $r = 10$.

2.1.3 Calcolo dell'orientamento

Il passo finale prima della fase di *description* è quello di assegnare un'orientamento ai *keypoint* estratti. Questo viene fatto per ottenere di rendere i *keypoint* **invarianti per rotazione**.

Per assegnare l'orientamento Lowe (ibid., p. 13) propone il seguente approccio: si seleziona l'immagine L, dalla piramide Gaussiana, che ha la scala più vicina a quella del punto a cui vogliamo assegnare un orientamento. In questo modo i calcoli sono effettuati in modo **invariante per scala**. Successivamente per ogni punto (x, y) in un area attorno al *keypoint*, area di

dimensione proporzionale alla sua scala, si calcolano magnitudine $m(x, y)$ e orientamento $\theta(x, y)$ del punto nel modo seguente:

$$m(x,y) = \sqrt{(L(x+1,y) - L(x-1,y))^2 + (L(x,y+1) - L(x,y-1))^2}$$

$$\theta(x,y) = \tan^{-1}((L(x,y+1) - L(x,y-1))/(L(x+1,y) - L(x-1,y)))$$

Una volta ottenute le magnitudini e gli orientamenti dei vari punti, viene creato un istogramma che discretizza in 36 *bin* gli angoli possibili, in modo da coprire 360 gradi. In base all'orientamento del gradiente $\theta(x, y)$ viene aggiunta all'istogramma la magnitudine del punto pesata da una finestra Gaussiana con σ pari a 1.5 volte la scala del punto.
Una volta creato l'istogramma in questo modo, gli orientamenti assegnati al *keypoint* sono quelli dominanti, cioè quelli che hanno un valore compreso fra quello del picco massimo e un 80% del picco massimo. Tuttavia dal momento che è stata effettuata una discretizzazione, i picchi ottenuti avranno valori interi ed è facile vedere che vi è una grossa perdita di informazioni. Quello che occorre fare allora è interpolare i picchi con una parabola, usando i 3 valori più vicini, per ottenere un approssimazioni migliore.
Si è parlato fino ad ora di picchi al plurale, ciò è dovuto al fatto che può capitare che vi siano più orientamenti dominanti. Se così dovesse accadere, semplicemente verrebbero aggiunti più *keypoint* con valori identici di posizione nell'immagine e nelle scale, ma orientamenti differenti.

2.1.4 Riepilogo

È stato mostrato come **SIFT** crea la **rappresentazione spazio-scala** di un'immagine e come vengono trovati gli estremi nello **spazio-scala**, in modo da essere **invarianti per cambi di scala**. È stato presentato il metodo con cui **SIFT** filtra i *keypoint* ottenuti, eliminando quelli con poco contrasto e quelli situati sui bordi degli oggetti. Infine è stato descritto in che modo **SIFT** assegna un orientamento ai **keypoint**, in modo da rendere possibile l'**invarianza per cambi di orientamento**.

2.2 Fase di description

Nella fase di *description* l'obiettivo è quello di creare dei **descrittori**, uno per ogni *keypoint*, che permettano di ottenere **invarianza per cambi di illuminazione** e **cambi di punto di vista**.
I **descrittori** creati con l'approccio di Lowe (Lowe 2004, pp. 14-16) non sono altro che istogrammi campionati in aree attorno ai *keypoint* che vengono inseriti in unico vettore con 128 *entry* finali (chiamato *feature vector*). Il campionamento avviene in modo del tutto simile a quello con cui venivano calcolati gli orientamenti.
Come in precedenza, viene scelta l'immagine più vicina della piramide Gaussiana usando la scala del *keypoint*. Successivamente si campionano 16 istogrammi da altrettante aree, distribuite in una griglia 4×4 attorno al *keypoint*; gli istogrammi sono discretizzati in modo da contenere 8 *bin* ciascuno ($4 \times 4 \times 8 = 128$). Per ottenere **invarianza per rotazione**, gli orientamenti e le coordinate dei punti che vengono inseriti nei *bin*, sono ruotati usando

l'orientamento del *keypoint* calcolato in precedenza.
I valori inseriti nei *bin* degli istogrammi corrispondono alle magnitudini dei punti pesate da una finestra Gaussian la cui σ è 1.5 volte quella del *keypoint*. Inoltre i valori sono anche interpolati trilinearmente per aumentare la precisione, quindi oltre a essere pesato ogni valore è anche moltiplicato per $(1 - d)$ per ogni dimensione (cioè x, y e σ), con d distanza del punto dal valore centrale del *bin* misurata nelle unità di spazio dell'istogramma.
Quello che si è ottenuto in questo modo è un *feature vector* che è **invariante per cambi di orientamento** ma che non è ancora **invariante per cambi di illuminazione**. Per far si che lo sia, Lowe (Lowe 2004, p. 16) propone di normalizzare a vettore unitario il *feature vector*. Questa scelta è motivata dal fatto che

> "un cambio nel contrasto dell'immagine in cui il valore di ogni pixel è moltiplicato da una costante moltiplicherà i gradienti per la stessa costante, questo cambio di contrasto sarà cancellato dalla normalizzazione del vettore."[5] (ibid., p. 16).

In questo modo il **descrittore** diventa, in generale, **invariante per cambiamenti di illuminazione affini**.
Si possono però anche verificare cambi di illuminazione non lineari, che

> "[...] possono causare grandi cambiamenti nelle magnitudini relative per alcuni gradienti, ma è meno probabile che interessino gli orientamenti dei gradienti."[6] (ibid., p. 16).

[5] "a change in image contrast in which each pixel value is multiplied by a constant will multiply gradients by the same constant, so this contrast change will be canceled by vector normalization." (trad. mia)
[6] "[...] can cause a large change in relative magnitudes for some gradients, but are less likely to affect the gradient orientations." (trad. mia)

Pertanto quello che viene fatto è applicare una soglia al *feature vector*, ossia ogni valore superiore a 0.2 (determinato sperimentalmente da Lowe) è posto uguale a 0.2. Questo per far si che valori i più abbiano meno peso. Il vettore è poi ri-normalizzato a vettore unitario.

2.2.1 Riepilogo

È stata descritta la fase di *detection*, cioè la fase in cui **SIFT** crea i **SIFT descriptor**, che sono dei vettori di 128 elementi. Questi permettono di descrivere in modo **invariante per rotazione**, per **cambi di illuminazione** e **punto di vista** i *keypoint* estratti nella fase precedente, rendendo ad esempio possibile effettuare *matching* di *keypoint* in modo molto più affidabile.

2.3 Applicazione all'object recognition

Sebbene **SIFT** sia un algoritmo che può essere usato per varie applicazioni, Lowe (Lowe 2004, p. 19-23) ha presentato un esempio di *object recognition*, che è importante per capire *come* possa essere utilizzato e mostra una delle sue applicazioni più note. Questa applicazione è particolarmente significativa perchè illustra un algoritmo per effettuare *matching* efficiente di *keypoint*, cosa utile in qualsiasi tipo applicazione, e che è diventato quasi parte di **SIFT** stesso.

2.3.1 Definizione di match

Si supponga di aver costruito un database che contiene **descrittori** di *keypoint* estratti da immagini di *training*, ad esempio immagini di oggetti di cui si vuole effettuare il riconoscimento.
Un *keypoint* in questo database è considerato un *match* per un dato *keypoint*, estratto dall'immagine di cui vogliamo fare il *match* col database, se è il suo ***nearest neighbor***, cioè se la **distanza Euclidea** fra il suo descrittore e il descrittore del *keypoint* è minima.
Un problema che emerge da questo approccio è che molte *feature* non avranno nessun *match* corretto nel database, ad esempio perchè non erano state identificate nelle immagini di *training*. Quello che Lowe (Lowe 2004, p. 20) propone per risolvere questo problema è paragonare la distanza dal secondo *nearest neighbor* che si sa provenire da un oggetto diverso dal primo con quella dal primo. Questo funziona perchè

> "*match* corretti devono avere il *neighbor* più vicino ad una distanza molto minore di quella del *match* incorretto più vicino al fine di ottenere un *matching* affidabile"[7] (ibid., p. 20).

In particolare Lowe scarta *match* la cui proporzione fra il primo *nearest neighbor* e il secondo supera 0.8, valore determinato sperimentalmente.

[7] "correct matches need to have the closest neighbor significantly closer than the closest incorrect match to achieve reliable matching." (trad. mia)

2.3.2 Individuazione di match

Quello di trovare il *nearest neighbor* di un dato punto è un problema complesso. Per punti con tante dimensioni (**SIFT** ne ha 128!) generalmente non è risolvibile in modo esatto in un tempo che si sa essere minore di quello della ricerca esaustiva. Questo fenomeno è noto come "*curse of dimensionality*". Ci sono tuttavia algoritmi, come quello che usa *k-d tree* di Friedman *et. al* (Friedman, Bentley e Finkel 1977), che permottono di effettuare *match* in un tempo piuttosto veloce e cioè, almeno in teoria, logaritmico nel numero di punti (ibid., 1977, p. 3).
Lowe (Lowe 2004, p. 20, 21) propone un algoritmo chiamato **Best-Bin-First** sviluppato da Beis e Lowe (Beis e Lowe 1997) e basato sempre su *k-d tree*, ma che usa un ordine di ricerca modificato. Prima di descrivere questo algoritmo è importante però capire cosa è un *k-d tree* e può aiutare in questo compito.

k-d tree e ricerca del nearest neighbor

k-d tree sta per albero k-dimensionale e, nonostante il suo nome, è un albero binario che viene usato per effettuare la ricerca di punti a più dimensioni. Ogni nodo dell'albero è infatti un punto k-dimensionale.
La semantica di questa struttura dati è la seguente: ogni livello ha quella che si può definire una "dimensione dominante", che è scelta ciclando fra le dimensioni dei punti (ad esempio se si rappresentano punti in 3D, il primo livello avra come "dimensione dominante" x, il secondo y, il terzo z, il quarto x e così via). La "dimensione dominante" indica come i nodi non foglia dividono l'iperpiano, infatti ogni nodo non foglia lo divide in due semispazi in modo perpendicolare alla "dimensione dominante". Quello che si verifica

allora è che i nodi del sotto-albero sinistro rappresentino i punti del semispazio sinistro mentre i nodi del sotto-albero destro quelli del semispazio destro. Una descrizione informale di come avviene la ricerca è la seguente: si inizia dalla radice dell'albero e ci si muove a destra o sinistra in base al risultato del confronto fra il valore del nodo e del *query point* nella "dimensione dominante" (se il *query point* ha un valore inferiore o uguale allora si va a sx, altrimenti a dx). Una volta raggiunto un nodo foglia ci si ferma e si prende quel nodo come ***nearest neighbor***. Successivamente si ripercorre l'albero partendo dal nodo foglia e andando verso la radice. A ogni nodo non foglia che si incontra si controlla se è più vicino del nodo migliore ed eventualmente lo si aggiorna. Poi si verifica se ci si potrebbero essere nodi nel semispazio non visitato più vicini del nodo migliore: se questo è il caso, si visita anche il sotto-albero corrispondente ripartendo dall'inizio dell'algoritmo altrimenti lo si scarta. La ricerca termina quando si giunge alla radice.

Best-Bin-First

L'algoritmo proposto da Lowe nasce da una considerazione sul modo in cui la ricerca del ***nearest neighbor*** viene effettuata nel *k-d tree*, infatti

> "[...] una larga parte di questa ricerca è passata esaminando *bin* nei quali solo una piccola frazione del loro volume potrebbe fornire il *nearest neighbour* [...]." [8] (Beis e Lowe 1997, p. 4).

Per risolvere questo problema sono presentate due idee (Beis e Lowe, 1997, p. 4): la prima è quella di stabilire un numero massimo di nodi foglia da visitare. Una volta visitato questo numero ci si ferma e quello che si è ottenuto

[8] "[...] a great deal of this search is spent examining bins in which only a small fraction of their volume could possibly supply the nearest neighbour [...]." (trad. mia)

è un *nearest neighbor* approssimato. La seconda è quella di usare un ordine di ricerca basato sulla distanza dal *query point*. Questo è implementabile tramite una coda con priorità in cui vengono inserite, ogni volta che si prende una decisione di andare in una certa direzione, *entry* corrispondenti alla scelta non effettuata (Beis e Lowe 1997, p. 4, 5).

2.3.3 Trasformate di Hough e verifica

Fino ad ora è stato mostrato come è possibile estrarre in modo efficiente i *match* dei *keypoint* di una data immagine da un database, eliminando nel frattempo *match* incorretti usando un test sul rapporto delle distanze dal primo e secondo *nearest neighbor*.
Questo non è sempre sufficiente e pertanto Lowe (Lowe 2004, p. 21) propone un ulteriore meccanismo per verificare e rendere più affidabili i *match*: questo meccanismo prevede l'uso della **Trasformata di Hough**, un metodo che permette a dati di "votare" per una certa interpretazione di un modello per poi estrarre l'interpretazione che ha ricevuto più voti. In particolare ogni *keypoint* vota per una certa posa di un oggetto usando le sue coordinate (x, y, σ) e il suo orientamento. In questo modo i *keypoint* vengono raggruppati in *bin* e quelli con almeno 3 *entry* sono soggetti ad un ulteriore verifica basata sul metodo dei minimi quadrati. Se passano questa verifica allora l'interpretazione da loro "sostenuta" è ritenuta corretta.

Capitolo 3

SIFT: un'implementazione Python

3.1 Python

La scelta del linguaggio per l'implementazione di **SIFT** è ricaduta su Python per vari motivi. Nonostante possa non sembrare adatto per implementare algoritmi di tipo numerico per via della sua lentezza, soprattutto se paragonato ad altri linguaggi come C, è bene tenere presente che vi sono molte librerie (ad esempio `numpy` e `scipy` per citarne qualcuna) che permettono di ottenere prestazioni molto buone per quanto riguarda calcoli di tipo scientifico.
Inoltre Python è un linguaggio con una sintassi estremamente chiara e che quindi si presta meglio di altri ad implementazioni semplici e in cui la logica degli algoritmi implementati non viene sommersa da sintassi a volte oscure e poco comprensibili.

L'implementazione è stata fatta in Python 2.7. Inoltre le seguenti librerie esterne sono state utilizzate: `cv2`, `matplotlib`, `numpy`, `scipy` e `json`. Non vi sono altri requisiti.

3.2 Scelte implementative

La principale scelta che è stata fatta è stata quella di rendere **SIFT** una classe, denominata `SiftDetector`, con un certo numero di strutture interne e metodi offerti per poter ottenere le sue funzionalità.
I metodi principali sono i seguenti:

1. `createPyramidAndDoG`: questo metodo prende in input un immagine, che può essere in scala di grigi o meno, e crea a partire da essa la **piramide Gaussiana** e in contemporanea la **piramide di DoG**.

2. `detectKeypoints`: come si può immaginare, questo metodo permette di ottenere le funzionalità di *keypoint detection* di **SIFT**.

3. `computeSIFTDescriptors`: usando questo metodo è possibile calcolare i **descrittori** se prima sono stati trovati dei *keypoint*.

Oltre a questi metodi la classe ne offre altri fra cui uno che permette di salvare i *keypoint* e **descrittori** calcolati, uno che permette di visualizzare le due piramidi e i *keypoint* trovati e uno che effettua il match fra due immagini. `SiftDetector` non implementa direttamente questi metodi ma utilizza altre classi che sono `FeaturesDB`, `SiftVisualizer` e `SiftMatcher` che si occupano rispettivamente di gestire database di *features*, visualizzare i dati di **SIFT** ed effettuare il *match* di *feature*.

3.3 Detection degli estremi dello spazio-scala

La parte di *detection* degli estremi è stata implementata seguendo quanto descritto nella sezione 3 da Lowe (Lowe 2004, pp. 5-10) del suo paper ed è composta da due parti.

3.3.1 Costruzione delle piramidi

La prima parte, quella di costruzione delle piramidi, è la seguente:

```
1 def createPyramidAndDoG(self, img):
2     MAX_OCT_NUM = int(round(log(min(img.shape), 2)))
3     self.TOT_OCTS = self.TOT_OCTS if self.TOT_OCTS > 0 else MAX_OCT_NUM
4     self.TOT_OCTS = self.TOT_OCTS - self.FST_OCT
5
6     if self.FST_OCT == -1:
7         img = resize(img, (0, 0), fx=2., fy=2., interpolation=INTER_LINEAR)
8         startingImageAssumedSigma = 1.
9     else:
10        startingImageAssumedSigma = .5
11
12    sigma = self.SIGMA_0
13    sigToApply = sqrt(sigma**2 - startingImageAssumedSigma**2)
14    img = GaussianBlur(img, (0, 0), sigToApply, sigToApply)
15
16    k = 2.**(1./self.TOT_SCLS)
17
18    L, D = [], []
19    self.pyramid = []
20    for i in xrange(self.TOT_OCTS):
21        for j in xrange(self.TOT_SCLS + 3):
22            if j == 0:
23                L.append(img)
24            else:
25                sigToObtain = sigma*(k**j)
26                sigToApply = sqrt(sigToObtain**2 - sigma**2)
27                L.append(GaussianBlur(img, (0, 0), sigToApply, sigToApply))
28                if j > 0:
29                    D.append(subtract(L[j], L[j - 1]))
30
```

```
31        sigma = 2*sigma
32        img = resize(L[self.TOT_SCLS],
                       (0, 0),
                       fx=.5,
                       fy=.5,
                       interpolation=INTER_NEAREST)
33        self.pyramid.append([L, D])
34        L, D = [], []
35
36    print 'Pyramid created'
```

Il codice funziona nel seguente modo: per prima cosa viene calcolato il massimo numero possibile di ottave che possono essere utilizzate, successivamente viene impostato il numero totale di ottave (`TOT_OCTS`) che può essere un valore positivo specificato dall'utente o appunto il massimo numero di ottave possibili. Il numero di ottave totali è anche dipendente dal valore della prima ottava (`FST_OCT`) che può essere 0 o -1. La semantica di questo valore è che l'utente può scegliere di raddoppiare la dimensione dell'immagine iniziale e in tal caso la variabile `FST_OCT` viene settata a -1.
Se necessario, allora, l'immagine viene raddoppiata usando la funzione `resize`.
Una volta raddoppiata l'immagine, viene creata la base della piramide sfocandola. La sfocatura applicata all'immagine tiene conto del fatto che si assume che l'immagine abbia una certa sfocatura di partenza (0.5 normalmente, 1.0 se è stata raddoppiata).
Se si applica una sfocatura Gaussiana, il cui valore è σ_{app}, ad un'immagine già sfocata, con sfocatura σ_{prec}, la sfocatura dell'immagine che si ottiene rispetterà la seguente equazione:

$$\sigma_{ott} = \sqrt{{\sigma_{app}}^2 + {\sigma_{prec}}^2}$$

Pertanto i valori delle sfocature da applicare, σ_{app}, sono calcolati nel modo seguente:

$$\sigma_{app} = \sqrt{{\sigma_{ott}}^2 - {\sigma_{prec}}^2}$$

Questo è ciò che viene fatto alla righe 13 e 25, 26.
Quello che viene fatto nei due cicli è la creazione della piramide vera e propria: L ad ogni iterazione è lista delle immagini sfocate per quell'ottava e D è la lista delle differenze di Gaussiane. Alla fine del ciclo interno, che serve a riempire L e D per quell'ottava, il valore di σ viene raddoppiato, viene presa dalla lista delle immagini sfocate l'immagine che ha sfocatura 2σ e viene posta come base della nuova piramide dopo essere stata dimezzata usando `resize`. L e D sono poi salvate in una struttura interna che contiene i vari livelle delle piramidi (`pyarmid`) e sono successivamente resettati.

3.3.2 Detection degli estremi

La *detection* vera e propria è invece implementata nel seguente metodo:

```
1  def detectKeypoints(self):
2    self.keypoints = []
3
4    if not self.pyramid:
5      exitWithErr('noPyramidErr', self.pyrerr)
6
7    computeMainOri = self.computeMainOrientations
8
9    for i in xrange(self.TOT_OCTS):
10     D = self.pyramid[i][1]
11     self.keypoints.append([])
12
13     for j in xrange(1, self.TOT_SCLS + 1):
14       height, width = D[j].shape
15
16       for y in xrange(1, height - 2):
17         for x in xrange(1, width - 2):
18           val = D[j][y][x]
19
20           isMax = val > 0 and self._isLocalExtremum(x, y, j, D,
                                                (lambda a, b: a >= b))
21           isMin = val < 0 and self._isLocalExtremum(x, y, j, D,
                                                (lambda a, b: a <= b))
```

```
22          if isMax or isMin:
23            interpolatedExtremum = self._filterLocalExtremum(x, y, j, D)
24
25            if interpolatedExtremum == None:
26              continue
27            else:
28              interpolatedExtremum.octave = i
29
30            mainOrientations = computeMainOri(interpolatedExtremum)
31            mainOriLen = len(mainOrientations)
32
33            for o in xrange(mainOriLen):
34              newOrientedKp = interpolatedExtremum.copy()
35              newOrientedKp.orientation = mainOrientations[o]
36              self.keypoints[i].append(newOrientedKp)
37
38  print 'Keypoints detected, filtered and orientated'
```

Innanzitutto `keypoints`, cioè la struttura dati che conterrà i *keypoint*, viene resettata e viene fatto un controllo sull'esistenza delle piramidi. Successivamente per ogni ottava della **piramide DoG** (cioè `D`), si aggiunge una nuova ottava vuota a `keypoints`.

Per ogni scala interna della **piramide DoG**, cioè dalla 1 alla s ($s + 2$ scale numerate da 0 a $s + 1$), si visita ogni punto, tranne quelli sui bordi, e si verifica se è un estremo con il metodo `_isLocalExtremum`. Si può notare come venga fatto un primo filtraggio degli estremi: se un valore positivo è un minimo o un valore negativo è un massimo, è perchè sono valori molto piccoli e che quindi con buona probabiltà verranno scartati durante il vero filtering dei punti (ad esempio perchè hanno un contrasto troppo basso).

Se il punto trovato è un canditato per essere un *keypoint* viene filtrato. Il metodo utilizzato è `_filterLocalExtremum`, che può ritornare `None`, se il filtraggio non è andato a buon fine, o un oggetto `Keypoint`. Infine vengono calcolati gli orientamenti dominanti del *keypoint* e per ogni orientamento viene aggiunto un *keypoint* a `keypoints`.

3.4 Interpolazione e filtraggio degli estremi

L'interpolazione, avviene nello stesso modo descritto da Lowe (Lowe 2004, pp. 10-12), tuttavia per assicurare la convergenza viene posta una soglia sul numero massimo di passi di interpolazione che vengono effettuati. Inoltre anzichè venire effettuata in un secondo momento, viene effettuata durante la fase di *detection* stessa. Questo sia per evitare di dover memorizzare troppi *keypoint* che tanto verrebbero scartati, sia per non dover scorrere due volte la lista di tutti i *keypoint* e quindi sprecare tempo.
Il filtraggio dei *keypoint* avviene subito dopo l'interpolazione, che è essa stessa una forma di filtraggio. Pertanto il filtraggio è implementato nello stesso metodo dell'interpolazione, cioè `_filterLocalExtremum`.

Il codice del metodo è il seguente:

```
1  def _filterLocalExtremum(self, x, y, scale, D):
2    interpolatedExtremum = Keypoint(x, y, scale)
3    interpolationWorked = False
4    args = [x, y, scale, D]
5    maxRows, maxCols = D[scale].shape
6
7    for i in xrange(self.MAX_INT_STEPS):
8      derivatives = array(self._computeDerivatives(*args))
9      hessian = array(self._computeHessian(*args))
10     hessDet = det(hessian)
11
12     retval, X = solve(hessian, derivatives, flags=DECOMP_LU)
13     if retval != True:
14       return None
15     X = map(lambda x: -x, X)
16
17     if abs(X[0]) < 0.5 and abs(X[1]) < 0.5 and abs(X[2]) < 0.5:
18       interpolationWorked = True
19       break
20
21     interpolatedExtremum.addOffset(X)
```

```
22      col, row, scl = interpolatedExtremum.getRoundedCoords()
23
24      if (col < self.BORDER_OFST or col >= maxCols - self.BORDER_OFST) or
           (row < self.BORDER_OFST or row >= maxRows - self.BORDER_OFST) or
           (scl < 1 or scl > self.TOT_SCLS):
25        return None
26      else:
27        args = [col, row, scl, D]
28
29      if not interpolationWorked:
30        return None
31
32      col, row, scl = interpolatedExtremum.getRoundedCoords()
33      extremumValue = D[scl][row][col]/255.
34
35      if abs((extremumValue + .5*dot(derivatives, X)))*self.TOT_SCLS <
           self.CONTR_THRESH:
36        return None
37
38      interpolatedExtremum.addOffset(X)
39      args = interpolatedExtremum.getRoundedCoords() + D
40      hessian = _computeHessian(*args)
41
42      hessDet = hessian[0][0]*hessian[1][1] - hessian[0][1]**2
43      hessTrace = hessian[0][0] + hessian[1][1]
44      if hessDet <= 0. or (hessTrace**2)/hessDet >=
           ((self.EDGE_THRESH + 1.)**2)/self.EDGE_THRESH:
45        return None
46
47      return interpolatedExtremum
```

Inizialmente viene creato un nuovo `Keypoint` usando posizione e scala fornite. Ad ogni passo di interpolazione vengono calcolate le derivate e l'Hessiana usando il metodo delle differenze finite, inoltre viene anche calcolato il determinante dell'Hessiana e viene risolto il sistema lineare composto da Hessiana e derivate, ottenendo così l'*offset* dal massimo. Si controlla poi se l'*offset* è inferiore a 0.5 in tutte le dimensioni. Se è così allora viene terminata l'interpolazione con successo, altrimenti l'*offset* è aggiunto al *keypoint* e si va al passo successivo.

Una volta terminata l'interpolazione si verifica se sia terminata con successo o meno. Se lo è allora avviene il *filtering* e quindi viene prima calcolato il valore

di D nell'estremo per controllare il contrasto del punto (`CONTRAST_THRESH` è 0.03, il minimo contrasto accettato). Questo valore è calcolato come

$$D(\hat{\mathbf{x}}) = D + \frac{1}{2}\frac{\partial D}{\partial \mathbf{x}}\hat{\mathbf{x}}$$

Successivamente viene controllato se l'estremo è un picco poco definito e pertanto viene ricalcolata l'Hessiana nell'estremo e quindi effettuato il test (`EDGE_THRESH` vale 10 ed è la soglia per i picchi poco definiti).

3.5 Calcolo dell'orientamento

Questa parte viene anch'essa effettuata durante la *detection* dei *keypoint*. Questo sempre per evitare di dover rivisitare la lista di tutti i *keypoint*.
Il calcolo degli orientamenti principali di un punto viene fatto chiamando il metodo interno `_computeMainOrientations`. Questo metodo semplicemente si occupa di definire alcuni valori per il calcolo dell'istogramma degli orientamenti, che viene fatto in modo generale da `_computeOrientationHistogram`, usato sia per il calcolo degli orientamenti sia per il calcolo dei descritori. Successivamente sono identificati i picchi dell'istogramma, che vengono poi interpolati e aggiunti alla lista degli orientamenti principali (`mainOrientations`). Di seguito il codice del metodo:

```
1  def _computeMainOrientations(self, kp):
2    mainOrientations = []
3    oriHistLen = 36
4    windowSigma = 1.5*self.SIGMA_0*(2**(kp.scale/self.TOT_SCLS))
5    nbrdRadius = int(round(3.*windowSigma))
6    factor = -1./(2*(windowSigma**2))
7
8    orientationHistogram = self._computeOrientationHistogram(kp,
```

```
9                                                    nbrdRadius,
10                                                 binsNum=oriHistLen,
11                                               windowSigma=windowSigma,
12                                             gaussianScaleFactor=factor)
13   margin = max(orientationHistogram)*self.PEAK_RATIO
14
15   for i in xrange(oriHistLen):
16     prec = i - 1 if i > 0 else oriHistLen - 1
17     succ = i + 1 if i < oriHistLen - 1 else 0
18     iethValue = orientationHistogram[i]
19     if iethValue >= margin and
20        iethValue > orientationHistogram[prec] and
21        iethValue > orientationHistogram[succ]:
22       ofst = self._interpolateHistogramPeak(orientationHistogram[prec],
                                                 orientationHistogram[i],
                                                 orientationHistogram[succ])
23       b = i + ofst
24       if b < 0.:
25         b += 36.
26       if b >= 36.:
27         b -= 36.
28       mainOrientations.append(b*10.)
29   return mainOrientations
```

Il metodo `_computeOrientationHistogram` funziona invece nel modo seguente: per prima cosa viene stavilito il rapporto fra numero di bin e angoli e cioè *bins* : 360, poi viene selezionata l'immagine sfocata più vicina alla scala del *keypoint*. Se si sta calcolando l'istogramma per il descrittore (`computingDescrHist = True`) allora vengono calcolati seno e coseno dell'orientamento del *keypoint*, che serviranno per ruotare i punti. Il doppio ciclo permette di visitare l'area attorno al *keypoint*. Se necessario le coordinate sono ruotate e poi, dopo un controllo sulla loro validità (cioè devono essere entro i limiti dell'immagine), vengono calcolati $L(x+1, y) - L(x-1, y)$ e $L(x, y+1) - L(x, y-1)$ (`xDiff` e `yDiff`) che sono poi usati per calcolare la magnitudine e l'orientamento come specificato da Lowe (Lowe 2004, p. 13). L'orientamento ottenuto va ruotato nel caso di calcolo di un descrittore, dopodichè si calcola il bin corrispondente e si aggiunge la magnitudine pesata all'istogramma, che alla fine è ritornato.

Il codice del metodo è questo:

```
1  def _computeOrientationHistogram(self, kp, nbrdRadius, binsNum, windowSigma,
                                    gaussianScaleFactor, computingDescrHist = False):
2    histogram = [0.] * binsNum
3    binsToDegreesRatio = binsNum/360.
4    x, y, scale = kp.getRoundedCoords()
5    L = self.pyramid[kp.octave][0][scale]
6    yUpperBound, xUpperBound = L.shape
7
8    if computingDescrHist:
9      cosOri, sinOri = cos(radians(kp.orientation)),
                        sin(radians(kp.orientation))
10
11   k = 0
12   for i in xrange(-nbrdRadius, nbrdRadius + 1):
13     for j in xrange(-nbrdRadius, nbrdRadius + 1):
14       if computingDescrHist:
15         i, j = i*cosOri - j*sinOri, i*sinOri + j*cosOri
16
17       xOfst, yOfst = int(round(x + i)), int(round(y + j))
18
19       if (xOfst + 1 >= xUpperBound) or
            (yOfst + 1 >= yUpperBound) or
            (xOfst - 1 < 0) or (yOfst - 1 < 0):
20         continue
21
22       xDiff = L[yOfst][xOfst + 1] - L[yOfst][xOfst - 1]
23       yDiff = L[yOfst + 1][xOfst] - L[yOfst - 1][xOfst]
24
25       weight = exp((i**2 + j**2)*gaussianScaleFactor)
26       magnitude = sqrt(xDiff**2 + yDiff**2)
27       orientation = fastAtan2(yDiff, xDiff)
28
29       if computingDescrHist:
30         orientation -= kp.orientation
31
32       binNum = round(orientation*binsToDegreesRatio)
33       if binNum >= binsNum:
34         binNum -= binsNum
35       elif binNum < 0:
36         binNum += binsNum
37
38       histogram[int(binNum)] += magnitude*weight
39       k += 1
40
41   return histogram
```

3.6 Calcolo dei descrittori

Il calcolo dei descrittori avviene calcolando, per ogni *keypoint*, i 16 istogrammi degli orientamenti centrati in altrettanti punti, che sono contenuti in una lista (`points`). In questo modo viene creato il *feature vector* con 128 dimensioni che viene normalizzato, limitato a 0.2 in ogni sua dimensione e poi ri-normalizzato. Il tutto è così implementato:

```
1  def computeSIFTDescriptors(self):
2    self.descriptors = []
3
4    if not self.keypoints:
5      print self.kperr
6      return
7
8    l = [-6, -2, 2, 6]
9    points = [[x, y] for x in l for y in l]
10
11   for o in xrange(len(self.keypoints)):
12   self.descriptors.append([])
13
14   appendNewDescriptorToCurrentOctave = self.descriptors[o].append
15   currKpsOctave = self.keypoints[o]
16
17   yMax, xMax = self.pyramid[o][0][0].shape
18   for k in xrange(len(currKpsOctave)):
19     self.descriptors[o].append([])
20
21     kp = currKpsOctave[k]
22     kpSigma = self.SIGMA_0*(2**(kp.scale/self.TOT_SCLS))
23     for p in points:
24       tmpKp = kp.copy()
25       tmpKp.x += p[0]
26       tmpKp.y += p[1]
27       self.descriptors[o][k] += self._computeOrientationHistogram(tmpKp,
28                                                                   nbrdRadius=2,
29                                                                      binsNum=8,
30                                                                  windowSigma=8,
31                                                 gaussianScaleFactor=(-1./16.),
32                                                       computingDescrHist=True)
33
34     self.descriptors[o][k] = normalize(self.descriptors[o][k])
35     self.descriptors[o][k] = map(lambda x: min(self.ILL_THRESH, x),
                                        self.descriptors[o][k])
36     self.descriptors[o][k] = normalize(self.descriptors[o][k])
37
38   print 'Descriptors computed'
```

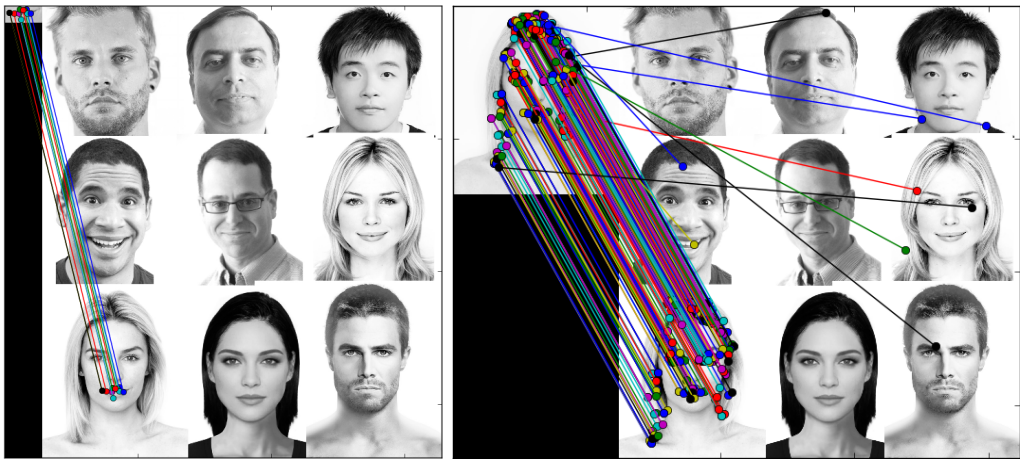

Figura 4: Un esempio di riconoscimento di volti (o di parti): a sx viene riconosciuta la bocca della ragazza in basso a sinistra, a dx viene riconosciuto tutto il volto.

3.7 Esempi di funzionamento

Verranno di seguito illustrati alcuni esempi di ciò che l'applicazione permette di fare, come il semplice riconoscimento di volti e oggetti. Verrà inoltre mostrato il modo in cui è possibile visualizzare i dati generati da **SIFT**, in particolare le piramidi e i *keypoint*.

3.7.1 Matching di volti e oggetti

Il *matching* è implementato in `Sift_Matcher` e avviene usando i *k-d tree* implementati in `scipy`. Quello che il *matcher* fa è caricare due database di descrittori, di cui uno è quello di cui si vogliono trovare i *match* e l'altro è quello contro cui il *match* è effettuato. Una volta caricati, crea un *k-d tree* che indicizza i descrittori contro cui fare *match* e per ogni descrittore di cui si vuole trovare il corrispondente, cerca il suo *nearest neighbor*

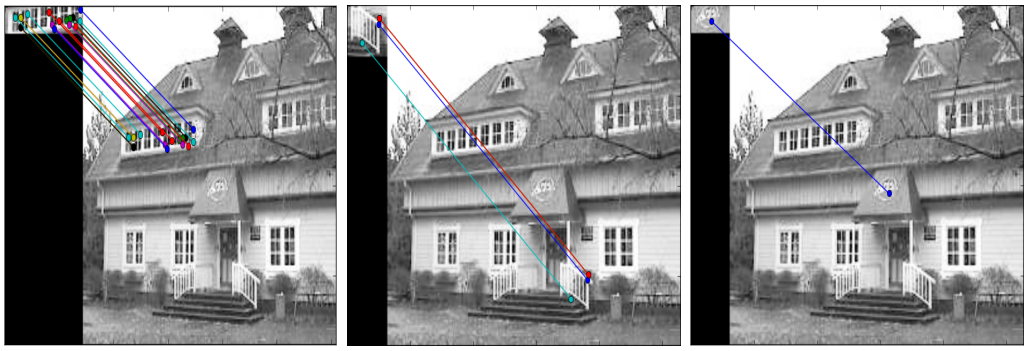

*Figura 5: In questa sequenza si può vedere come tre oggetti, o meglio parti della casa, vengano ritrovate nell'immagine generale usando **SIFT**.*

nell'albero. Trovato il *nearest neighbor* si memorizzano i *keypoint* che corrispondono ai descrittori che hanno fatto *match*. Dopo aver trovato tutte le corrispondenze viene disegnata un'immagine, unendo le due immagini a cui i descrittori appartenevano, con i *match* rappresentati tramite linee colorate fra due *keypoint*, come visibile nelle figure 4 e 5.

3.7.2 Visualizzazione delle strutture dati SIFT

Per quanto riguarda la visualizzazione delle strutture dati create da **SIFT**, la si può trovare implementata nella classe `SiftVisualizer`, che offre la possibilità di navigare le piramidi create da **SIFT** e di visualizzare semplicemente i *keypoint* sull'immagine corrispondente.
La navigazione delle piramidi avviene tramite l'interfaccia offerta da `matplotlib` e si può effettuare usando la tastiera. In particolare i *keybindings* sono i seguenti: con W e S ci si può spostare verso livelli superiori o inferiori delle piramidi; usando A e D è possibile muoversi fra le scale ed eventualmente fra le ottave una volta raggiunto il limite di scale di una certa ottava; Q ed E effettuano invece il cambio fra le piramidi, in particolare il primo seleziona la

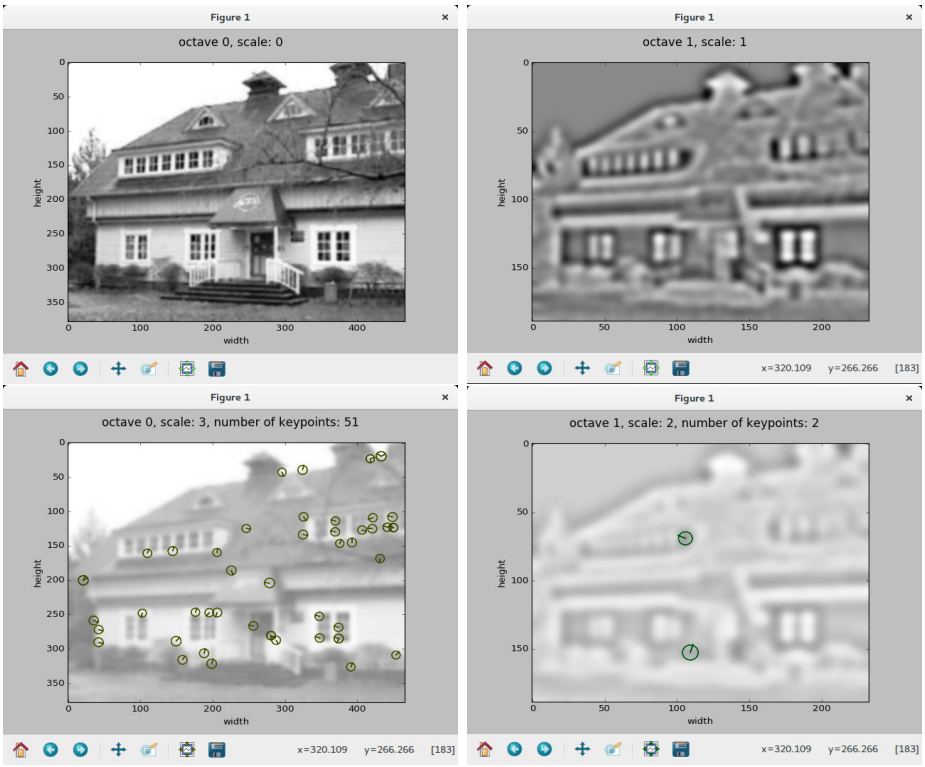

Figura 6: È possibile vedere quattro esempi di navigazione: in alto sono visibili due livelli delle piramidi, quella ***Gaussiana*** *a sx e quella* ***DoG*** *a dx. In basso è possibile vedere come i keypoint trovati siano visualizzati sulle piramidi.*

piramide Gaussiana, se non è già selezionata, e il secondo seleziona quella **DoG**; infine con *F* è possibile selezionare e deselezionare la visualizzazione dei *keypoint*. Un esempio di tutto questo è visibile in figura 6.

È inoltre possibile visualizzare semplicemente i *keypoint* che sono stati trovati e visualizzare i *keypoint* prima del loro filtraggio o prima dell'assegnamento degli orientamenti, ad esempio per poter capire meglio che tipo di *keypoint* vengono eliminati o quanti ne vengono eliminati dal processo di *filtering*.

Conclusioni

Per concludere verrà effettuato un breve riepilogo di ciò che è stato visto. Nel primo capitolo è stata presentata la disciplina della *Computer Vision*, sono state elencate alcune delle sotto-discipline che la compongono ed è stato presentato come **SIFT** si inserisce in questo contesto. Sono stati inoltre introdotti alcuni termini, come quello di *feature*, che sono centrali alla discussione.

Successivamente si è passati a descrivere **SIFT** in un maggior grado di dettaglio e sono state presentate le due fasi principali di cui si compone, cioè quella di *detection* e quella di *description*. Sono state definite le caratteristiche che una *feature* deve possedere ed è stato definito il concetto di **spazio-scala**, introducendo quindi la struttura a piramide. È stato anche mostrato come **SIFT** effettua l'interpolazione e il *filtering* delle *feature* identificate e come viene assegnato loro un orientamento. Per quanto riguarda la fase di *description* è stato illustrato come avviene la computazione dei **descrittori**.

Oltre a queste due fasi è stato anche descritta un'applicazione comune di **SIFT** e cioè quella dell' *object recognition*. Questa applicazione ha permesso di vedere in che modo è possibile effettuare *match*, in modo efficiente ma accurato, fra *keypoint* di **SIFT**.

È stata infine presentata un'implementazione di *SIFT* rivolta all'*object recognition* svolta in Python. Sono state definite alcune scelte implementative

iniziali per poi passare alla descrizione dell'implementazione delle singole parti. Al termine di questa descrizione sono stati mostrati alcuni esempi di funzionamento dell'applicazione.

Lavoro futuro

Sebbene gli esempi presentati possano essere interessanti c'è ancora molto lavoro che può essere svolto, sia per quanto riguarda l'aggiunta di nuove funzionalità, sia per quanto riguarda il miglioramento generale dell'architettura dell'applicazione e della robustezza dell'implementazione.
In particolare sarebbe importante aggiungere il *matching* come descritto nella sezione 2.3, al fine di ottenere dei risultati più accurati. Inoltre si potrebbe migliorare l'attuale visualizzazione delle strutture dati e aggiungere la possibilità di visualizzarne di nuove, come ad esempio i **descrittori** dei *keypoint*, cioè gli istogrammi da cui sono stati creati. Sarebbe altrettanto importante migliorare i *keypoint* trovati, cioè renderli più stabili, aumentare il numero di *keypoint* trovati e i migliorare il calcolo dei **descrittori**.

Bibliografia

Bay, H. et al. (2008). "SURF: Speeded Up Robust Features". In: *Computer Vision and Image Understanding (CVIU)*. Vol. 110. 3, pp. 346–359.

Beis, J.S. e D.G. Lowe (1997). "Shape indexing using approximate nearest-neighbour search in high-dimensional spaces". In: *Conference on Computer Vision and Pattern Recognition*. Porto Rico, pp. 1000–1006.

Canny, J. (1986). "A Computational Approach to Edge Detection". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-8.6, pp. 679–698.

Friedman, J.H., J.L. Bentley e R.A. Finkel (1977). "An Algorithm for Finding Best Matches in Logarithmic Expected Time". In: *ACM Transactions on Mathematics Software* 3.3, pp. 209–226.

Harris, C. e M.J. Stephens (1988). "A combined corner and edge detector". In: *Alvet Vision Conference*, pp. 147–152.

Ikeuchi, K., Y. Matsushita e R. Kawakami (2014). "Preface". In: *Computer Vision - A Reference Guide*. A cura di K. Ikeuchi. Stati Uniti: Springer US, p. v.

Lindeberg, T. (1994). "Scale-space theory: A basic tool for analysing structures at different scales". In: *Journal of Applied Statistics* 21.2, pp. 224–270.

Lindeberg, T. (2008). "Scale-space". In: *Encyclopedia of Computer Science and Engineering*. A cura di B. Wah. Vol. IV. Hoboken, New Jersey: John Wiley e Sons, pp. 2495–2504.

Lowe, D.G. (1999). "Object recognition from local scale-invariant features". In: *International Conference on Computer Vision*. Corfu, Greece, pp. 1150–1157.

— (2004). "Distinctive image features from scale-invariant keypoints". In: *International Journal of Computer Vision* 60.2, pp. 91–110.

Mikolajczyk, K. e C. Schmid (2005). "A performance evaluation of local descriptors". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 27.10, pp. 1615–1630.