Università Politecnica delle Marche
Ingegneria Meccanica

Impiego della Visual Odometry per la rilevazione della posa di telecamera in sistemi di machine vision

Visual Odometry for camera pose estimation in machine vision systems

Tesi di laurea Triennale di:

Jacopo William Vernelli

Relatore:

Prof. Gian Marco Revel

Correlatori:

Ing. Nicola Giulietti

Ing. Paolo Chiariotti

Anno Accademico: 2019-2020

Sommario

In questa tesi si descrive come si possa ricavare quali siano l'orientamento e le dimensioni di una superficie attraverso l'odometria visuale.

Si discute, in particolare, un algoritmo appositamente creato in ambiente di sviluppo MatLab per stimare la posizione di quattro punti proiettati su una parete; come si vedrà più avanti, le coordinate di questi punti consentono di stabilire la posa del sistema di visione.

L'algoritmo sviluppato richiede due soli input: un video della superficie analizzata in cui la telecamera si sposta inquadrando la parete da diverse angolature e la misura della distanza (in linea d'aria) percorsa dalla camera durante la ripresa.

Dopo una breve descrizione dello stato dell'arte della misurazione di oggetti a distanza, si descrive quali sono le basi delle tecniche odometriche con cui il programma calcola la traiettoria percorsa durante la ripresa e la posizione degli oggetti inquadrati.

Ci si concentra poi sulle funzioni native di MatLab sfruttate dal programma e si illustrano alcuni test realizzati su quest'ultime. Infine si riporta l'algoritmo descrivendo la funzione di ogni sua sezione e sottoprogramma.

Indice

1. Introduzione

Per poter estrarre dati metrici da un'immagine, è necessario conoscere le posizioni nello spazio di quattro punti chiave visibili in essa. In questo capitolo si vanno ad analizzare i metodi più comuni con cui si possono ricavare le coordinate di un punto dalla distanza, ci si concentra sulla misura telemetrica piuttosto che su quella in loco perché l'algoritmo che si tratterà in seguito è stato pensato per applicazioni con soggetti difficilmente raggiungibili.

Tra gli strumenti più comuni nella telemetria troviamo le cosiddette *stazioni totali* che misurano le coordinate sferiche di un punto fornendo l'angolo azimutale e zenitale con cui un proiettore laser è orientato. Tale laser ha la funzione di misurare il modulo della distanza del punto, nel campo di applicazione del programma illustrato nella presente tesi le modalità principali con cui questo processo avviene sono due:

- Misura del tempo di volo
- Misura a modulazione di intensità

Se si sceglie di misurare il tempo di volo si invia un segnale laser e si misura il tempo che il raggio impiega per raggiungere l'obbiettivo, essere riflesso e tornare al mittente; trovato questo dato si può facilmente ricavare la distanza percorsa dal laser[1].

Col secondo metodo si usa invece un segnale laser la cui intensità varia in modo periodico, per esempio in modo sinusoidale. In questo caso la grandezza di output è lo sfasamento tra l'intensità del fascio in uscita e l'intensità che ritorna allo strumento dopo che il laser è stato riflesso dall'obbiettivo. Raccogliendo questo dato per un insieme di frequenze sufficientemente ampio si può determinare univocamente quale sia il numero di lunghezze d'onda tra lo strumento ed il bersaglio per ogni frequenza testata[1].

I telemetri laser a modulazione di intensità consentono accuratezze più elevate, mentre gli strumenti a tempo di volo raggiungono normalmente range di misura maggiori. Ciò scaturisce dal fatto che nelle stazioni a tempo di volo il laser varia la sua intensità in modo impulsivo e quindi questi raggi hanno una maggiore intensità istantanea a parità di energia emessa, consentendo al laser di mantenersi leggibile dal rilevatore anche dopo aver percorso distanze maggiori di quelle permesse da un telemetro a sfasamento[1].

Questo vantaggio permette ai telemetri ad impulsi di poter misurare superfici non-cooperanti con gradi in inclinazione maggiori rispetto alle controparti modulate per intensità.

Quando si utilizza una stazione laser si deve infatti tener conto del tipo di superficie a cui il punto da misurare appartiene, esse si dicono:

- Riflettenti: se riflettono un raggio luminoso in una precisa direzione.
- Diffusive: se il raggio incidente viene riflesso in tutte le direzioni con intensità variabile a seconda dell'angolo di riflessione.

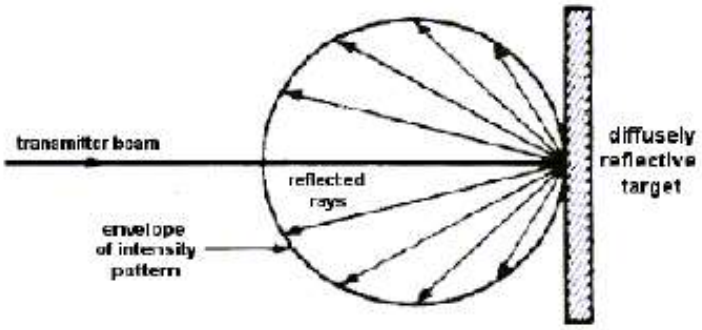

Figura 1: Raggio luminoso incidente su superficie non-cooperante

Quando si sceglie il luogo da cui effettuare la misurazione è quindi necessario verificare la fattibilità della stessa in funzione del tipo di superficie e potrebbe essere necessario avvalersi di *corner cube retroreflector* per far tornare il laser con la stessa direzione qualunque sia l'angolo di ingresso[1].

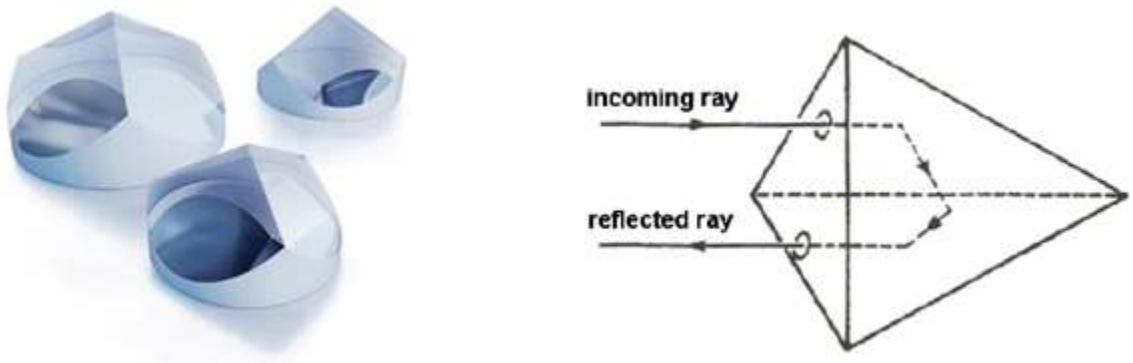

Figura 2: Immagine e schema di funzionamento dei corner cube

Per evitare di dover raggiungere il punto da misurare per installare un riflettore, si può utilizzare la fotogrammetria per ricavare le coordinate di un oggetto a prescindere dall'orientamento della superficie.

Si usa in questo caso una coppia di fotocamere di cui sono noti posizione e orientamento relativo per triangolare le coordinate di un punto nello spazio. Questo metodo richiede all'utilizzatore di indicare il punto chiave da determinare in entrambe le immagini, esistono algoritmi che svolgono questa funzione ma, nel caso in cui le immagini non presentino feature facilmente identificabili,

potrebbe essere necessario applicare sulla superficie target dei bersagli riconoscibili (**Figure 3Figure 3**) da una macchina.

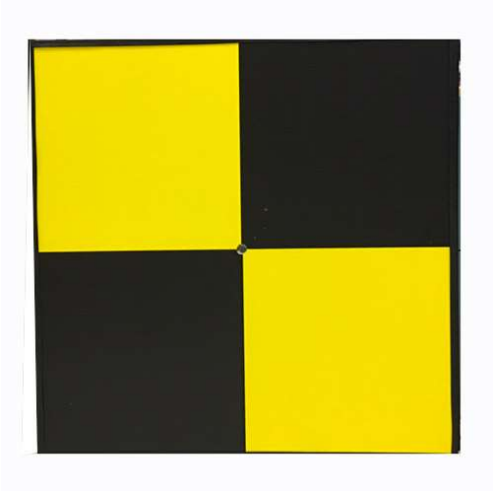

Figure 3: Tipico bersaglio per fotogrammetria

All'interno di questa tesi si utilizza una variante di questa tecnica in cui le immagini vengono catturate da una sola fotocamera in movimento rispetto ad un soggetto fisso: non si hanno in questo caso informazioni sulle pose da cui vengono scattate le immagini, ma l'unico input metrico è la distanza in linea d'aria tra la posizione iniziale e quella finale della ripresa. Il capitolo successivo descrive come si possono triangolare le coordinate di punti con questi dati.

Fondamenti di fotogrammetria

La fotogrammetria è la tecnica che permette di ricavare i dati metrici di un oggetto confrontando una coppia di fotogrammi rappresentanti lo stesso soggetto, realizzati da posizioni ed angolature differenti.
Il presente lavoro si inquadra in un ambito più ampio finalizzato all'identificazione di cricche superficiali in strutture in calcestruzzo. Per poter valutare in maniera oggettiva l'estensione delle cricche a partire da un'immagine, si rende necessario effettuare una compensazione degli effetti prospettici attraverso la conoscenza della posa relativa tra il sistema di visione e la superficie target, nonché la distanza di un punto tra detti sistemi. In questo modo sarà possibile costruire una rappresentazione ortogonale del piano dalla quale sarà possibile misurare le dimensioni cercate. Tale processo è denominato *raddrizzamento fotografico*.

Raddrizzamento fotografico

I punti dell'oggetto sono legati ai punti dell'immagine da una corrispondenza biunivoca detta omografia. Se:

- X e Y sono le coordinate di un punto sull'oggetto;
- x e y sono le coordinate dell'immagine del punto sul fotogramma;
- a_1, a_2, a_3, b_1, b_2, b_3, c_1, c_2 sono i parametri che definiscono la trasformazione omografica;

L'omografia è esprimibile tramite le equazioni[2]:

$$X = \frac{a_1x+a_2y+a_3}{c_1x+c_2y+1} \qquad X = \frac{b_1x+b_2y+b_3}{c_1x+c_2y+1}$$

Per calcolare gli 8 parametri che definiscono la trasformazione omografica è quindi necessario conoscere le coordinate di almeno 4 punti dell'oggetto, espresse nel sistema di riferimento oggetto e individuabili nel sistema di riferimento immagine. Tali punti devono essere opportunamente distribuiti sull'immagine.
In questo modo si hanno a disposizione 4x2 = 8 elementi per definire gli 8 parametri incogniti. È infatti possibile scrivere otto equazioni per la determinazione delle otto incognite.
Una volta determinati i parametri dell'omografia che lega l'immagine alla parete reale è possibile ricavare le coordinate di un qualsiasi punto visibile dall'immagine grazie alle due formule precedenti.

Modello *pin-hole* e sistemi di riferimento utilizzati

Si trattano le telecamere attraverso il modello *pin-hole:* le deviazioni che le lenti impongono ai raggi di luce prima di raggiungere il sensore (o la pellicola, se si tratta di fotocamera analogica) vengono schematizzati semplicemente con un punto, chiamato *centro di proiezione*, ed un *piano d'immagine*, rappresentante la superficie del sensore.

Si immagini che ogni pixel sul piano d'immagine sia collegato attraverso una retta al punto nello spazio che esso rappresenta, si definisce *centro di proiezione* il punto in cui si incontrerebbero tutte le rette.

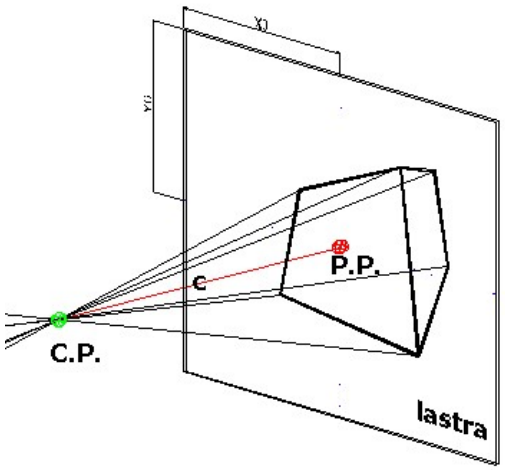

Figura 4: Modello pin-hole

Questo modello non tiene conto delle distorsioni provocate dalle lenti, si vedrà più avanti come si possa tentare di compensare tale fenomeno applicando una trasformazione all'immagine catturata in funzione della distanza focale. Tale processo ha l'obbiettivo di produrre un'immagine che si avvicina il più possibile a ciò che si otterrebbe se la fotocamera funzionasse esattamente secondo il modello *pin-hole*.

Bisogna ricordare come queste distorsioni sono accentuate vicino ai bordi dell'immagine e diminuiscono spostandosi verso il centro, fino ad annullarsi nel punto principale; Può quindi essere utile mantenere il soggetto analizzato al centro dell'immagine per diminuire l'errore.

Nella fotogrammetria si può trattare un proiettore come se fosse una fotocamera poiché anche ad esso può essere applicato il modello *pin-hole*. In questo caso è l'operatore a scegliere le coordinate sul piano immagine impostando il pattern da riprodurre; dato che il proiettore contiene un insieme di lenti simili a quelle di una videocamera si può assegnare anche ad esse un centro di proiezione ad una data distanza focale dal piano immagine. Si può quindi immaginare che i raggi luminosi generati dal proiettore partano da un insieme di punti sul piano immagine, passino attraverso il centro di proiezione e proseguano fino al primo oggetto opaco che incontrano.

Dato che sia i proiettori che le fotocamere hanno la funzione di mettere in relazione una serie di coordinate mondo tridimensionali con delle coordinate immagine bidimensionali, essi posso essere utilizzati in modo equivalente[3].

Prima di iniziare la trattazione del modello proiettivo, è bene introdurre come l'i-esimo punto può essere rappresentato nei vari sistemi di riferimento mostrati in **Figura 5** che verranno presi in esame:

1. Coordinate Immagine (2D in pixel): $u_i = (u_i , v_i)^T \in R^2$;
2. Coordinate Sensore (2D in mm.): $\tilde{u}_i = (\tilde{u}_i , \tilde{v}_i)^T \in R^2$;
3. Coordinate Camera (3D in mm.): $p_{Ci} = (\tilde{x}_i , \tilde{y}_i , \tilde{z}_i)^T \in R^3$;
4. Coordinate Mondo (3D in mm.): $p_{Wi} = (x_i , y_i , z_i)^T \in R^3$.

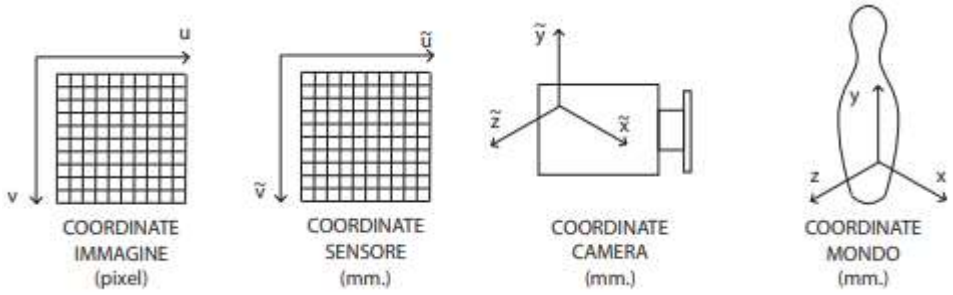

Figura 5: Sistemi di riferimento

Triangolazione di un punto nello spazio

Con questa serie di passaggi di coordinate consistente in ridimensionamenti, rotazioni e traslazioni siamo in grado di instaurare una relazione diretta tra un punto nel mondo esterno (misurato quindi in millimetri) ed un punto in un'immagine (misurato in pixel).

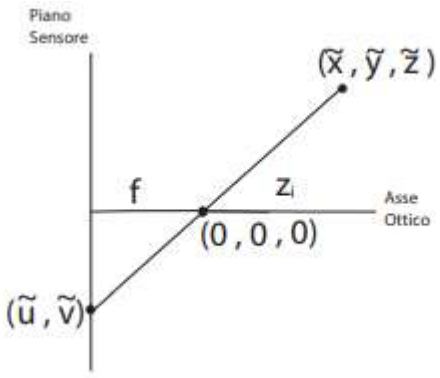

Figura 6: Proiezione di un punto

In **Figura 6** è rappresentato in maniera schematica il processo di formazione delle immagini su un sensore. Si vede subito come i rapporti dei triangoli generati dai raggi ottici mettono in relazione un punto p_C con un punto $\tilde{u}$. Si ha infatti:

$$(\tilde{u}_i , \tilde{v}_i)^T = f / \tilde{z}_i * (\tilde{x}_i , \tilde{y}_i)^T \quad (1)$$

dove f indica la distanza focale.

Per passare dalle coordinate sensore alle coordinate immagine è necessario applicare la trasformazione.

$$(u_i , v_i)^T = (D_u*\tilde{u}_i , D_v*\tilde{v}_i)^T + (u_0 , v_0)^T \quad (2)$$

in cui le coordinate $(u_0 , v_0)^T$, corrispondenti al punto principale, tengono conto della variazione dell'origine delle coordinate nell'immagine acquisita rispetto alla proiezione del punto focale del sensore. I valori D_u e D_v sono dei fattori di conversione [pixel/metro] tra le unità di misura dei sistemi di riferimento e tendenzialmente accade che $D_u = D_v$. Risulta sconveniente tuttavia parlare di focali e fattori di conversione e per questo motivo si preferisce utilizzare le grandezze k_u e k_v, chiamate lunghezze focali efficaci, misurate in pixel e definite come:

$$k_u = D_u*f \text{ e } k_v = D_v*f \quad (3)$$

In presenza di un'ottica simmetrica si ha che $k_u = k_v$. A causa del rapporto presente, l'equazione non è rappresentabile sotto forma di sistema lineare. Tuttavia, ricorrendo alle coordinate omogenee in cui si aggiunge una variabile ed un vincolo ulteriore, il sistema si può scrivere come:

$$(\lambda*u_i , \lambda*v_i , \lambda)^T = \lambda * (u_i , v_i , 1)^T = \mathbf{K} * (\tilde{x}_i , \tilde{y}_i , \tilde{z}_i)^T \quad (4)$$

e va risolto per $\lambda = z_i$. È questo il motivo per cui λ viene sottinteso a favore delle coordinate omogenee. Per tornare alle coordinate non omogenee basta dividere le prime due coordinate per la terza ed ottenere così il sistema (1).

In sostanza l'utilizzo delle coordinate omogenee permette di rendere implicita la divisione per la coordinata z. Unendo le equazioni (1), (2), (3) e (4) possiamo riscrive la matrice **K** come:

$$K = \begin{bmatrix} D_u f & k_\gamma & u_0 \\ 0 & D_v f & v_0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} k_u & k_\gamma & u_0 \\ 0 & k_v & v_0 \\ 0 & 0 & 1 \end{bmatrix} \quad (5)$$

La matrice **K**, chiamata *Matrice dei Parametri Intrinseci*, è una matrice triangolare superiore composta da 5 parametri. Il parametro k_γ, detto *skew factor*, in sostanza tiene conto dell'angolo tra il sensore ed il sistema di riferimento della camera. Nei moderni sistemi si ha $k_\gamma = 0$. La conoscenza di questa matrice permette la trasformazione di linee e piani dalle coordinate camera alle coordinate immagine e di punti e linee dalle coordinate immagine alle coordinate camera al netto dei disturbi distorsivi.

Per passare dalle coordinate mondo alle coordinate camera vanno prima fatte alcune supposizioni:

1. Per ottenere il sistema di riferimento finale, gli assi devono essere scambiati tramite una trasformazione $\mathbf{\Pi}$;

2. La camera può essere ruotata tramite una trasformazione $\mathbf{R_{bw}}$. Dopo ciò i 2 sistemi di riferimento potrebbero non coincidere;

3. Il punto pin-hole non coincide con il punto $(0, 0, 0)^T$ ma è posto in un punto $t_0 = (x_0, y_0, z_0)^T$ espresso in coordinate mondo.

La trasformazione tra coordinate mondo e coordinate camera risulta essere quindi la somma di 2 rotazioni e di una traslazione. La rotazione complessiva risulta essere $\mathbf{R} = \mathbf{R_{wc}} = \mathbf{\Pi}\ \mathbf{R_{bw}}^{-1}$.

Va ricordato che le matrici di rotazione sono ortonormali e quindi la matrice inversa è uguale alla matrice trasposta.

Sia quindi $(x_i , y_i , z_i)^T$ un punto in coordinate mondo e sia $(\tilde{x}_i , \tilde{y}_i , \tilde{z}_i)^T$ un punto in coordinate camera, la relazione che li lega è definita come:

$$\begin{bmatrix} \tilde{x}_i \\ \tilde{y}_i \\ \tilde{z}_i \end{bmatrix} = \boldsymbol{R}\left(\begin{bmatrix} x_i \\ y_i \\ z_i \end{bmatrix} - \boldsymbol{t_0}\right) = \boldsymbol{R}\begin{bmatrix} x_i \\ y_i \\ z_i \end{bmatrix} - \boldsymbol{R}\,\boldsymbol{t_0} \tag{6}$$

dove **R** è una matrice 3×3, chiamata Matrice di Rotazione, che tiene conto anche della variazione dei segni degli assi oltre al cambiamento del sistema di riferimento. Il vettore t_0 invece tiene conto della traslazione del punto *pin-hole* in coordinate camera.

Ricorrendo ancora una volta alle coordinate omogenee è possibile accorpare in una forma più semplice la matrice **R** ed il vettore t_0 in una matrice 3×4. Grazie a questa trasformazione è possibile scrivere la relazione che lega in coordinate omogenee un punto in coordinate mondo ad un punto in coordinate immagine:

$$\lambda * (u_i , v_i , 1)^T = \mathbf{K}\ [\mathbf{R}|\tilde{\boldsymbol{t}}_\mathbf{0}]\ (x_i , y_i , z_i , 1)^T \text{ con } \tilde{\boldsymbol{t}}_\mathbf{0} = -\mathbf{R}\ \mathbf{t_0} \text{ e } \lambda \in R \tag{7}$$

con $[\mathbf{R}|\tilde{\mathbf{t}}_\mathbf{0}]$ detta Matrice dei Parametri Estrinseci.

A meno di un fattore di scala λ è possibile quindi definire l'Equazione Finale della Pin-Hole Camera come:

$$(u_i , v_i , 1)^T = \mathbf{K}\ [\mathbf{R}|\tilde{\mathbf{t}}_\mathbf{0}]\ (x_i , y_i , z_i , 1)^T \tag{8}$$

Il Sistema 8 è un risultato di notevole importanza in quanto, a meno di un fattore di scala e di distorsioni dovuti alla lente, instaura una relazione diretta e univoca tra i punti in coordinate mondo e i punti nelle coordinate immagine. La relazione inversa che lega invece i punti delle coordinate immagini ai punti delle coordinate mondo è:

$$(x_i , y_i , z_i)^T = \lambda * \mathbf{R}^{-1}\ \mathbf{K}^{-1}\ (u_i , v_i , 1)^T + \mathbf{t_0} \tag{9}$$

Avendo a disposizione due fotocamere chiamate 1 e 2, è possibile usare l'equazione (9) per confrontare le coordinate $(u_{i1} , v_{i1} , 1)^T$ e $(u_{i2} , v_{i2} , 1)^T$ dello stesso punto i:

$$\lambda_1 * \mathbf{R_1}^{-1}\ \mathbf{K_1}^{-1}\ (u_{i1} , v_{i1} , 1)^T + \mathbf{t_{01}} = (x_i , y_i , z_i)^T = \lambda_2 * \mathbf{R_2}^{-1}\ \mathbf{K_2}^{-1}\ (u_{i2} , v_{i2} , 1)^T + \mathbf{t_{02}}. \tag{10}$$

Posto il sistema di riferimento assoluto come coincidente a quello della camera 1 ($\mathbf{R_1}^{-1} = \mathbf{I}$, $\mathbf{t_{01}} = \mathbf{0}$) e noto l'orientamento e la posizione relativa della camera 2 (quindi la matrice $\mathbf{R_2}^{-1}$ ed il vettore $\mathbf{t_{02}}$) oltre alle matrici dei parametri intrinseci $\mathbf{K_1}$ e $\mathbf{K_2}$ è possibile risolvere il sistema (10) per le incognite λ_1 e λ_2. Una volta ricavata $\lambda_1 = z_i$ è possibile trovare le due coordinate rimanenti con l'equazione (1)

in modo da determinare la posizione del punto i nello spazio rispetto al sistema di riferimento della telecamera 1.

Calcolo dell'orientamento relativo delle due fotocamere

Laddove non fosse possibile conoscere l'orientamento e la posizione relativa tra le due camere è comunque possibile ricavarle a partire dalla proiezione di 8 punti sull'oggetto tramite il calcolo della matrice fondamentale e della matrice essenziale.

La matrice essenziale è lo strumento fondamentale capace di incorporare tutti i vincoli geometrici dati dalla ripresa di due immagini. Dalla relazione che collega le coordinate immagine X_1, $X_2 \in R^3$ riferite allo stesso punto P rispetto a due differenti telecamere

$$X_2 = \mathbf{R}\, X_1 + \mathbf{T}. \tag{11}$$

si può dimostrare il seguente teorema[4]:

Teorema. Siano x_1, x_2 immagini dello stesso punto P secondo due telecamere con posizione relativa ($\mathbf{R}$, $\mathbf{T}$) $\in$ SE(3) allora x_1, x_2 soddisfano l'equazione

$$x_2^T\, \mathbf{T}\hat{}\, \mathbf{R}\, x_1 = 0 \text{ , dove } \mathbf{T}\hat{} \in \mathcal{M}_3(R) \mid \mathbf{T}\hat{}\, u = \mathbf{T} \times u \;\forall u \in R^3 \tag{12}$$

Si può ora definire come matrice essenziale la matrice $\mathbf{E} = \mathbf{T}\hat{}\, \mathbf{R} \in \mathcal{M}_3(R)$

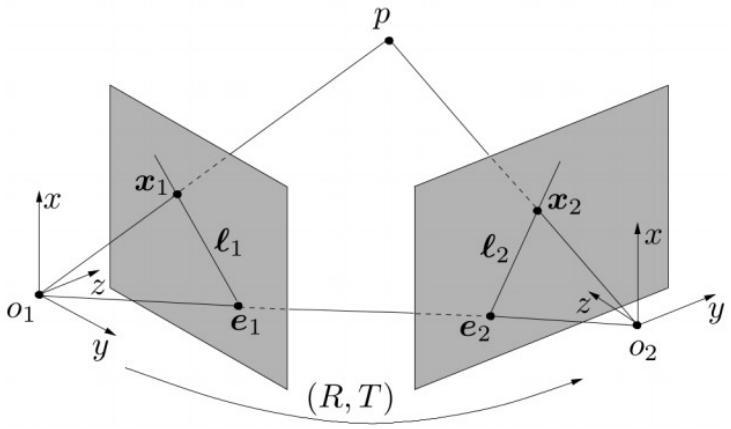

Figura 7: Triangolazione con due fotocamere

Come dimostrato in [4], una volta ricavata la matrice essenziale si può determinare in maniera univoca la matrice rotazione **R** ed il vettore traslazione **T** che la compongono.

L'interesse per la matrice essenziale deriva dal fatto che questa è direttamente calcolabile a partire dalle corrispondenze di punti dati. Infatti, come si è visto, ogni corrispondenza di punti si traduce in un'equazione del tipo

$$x_2^T\, \mathbf{E}\, x_1 = 0 \tag{13}$$

che è lineare rispetto agli elementi di **E**. Dunque, sono sufficienti otto corrispondenze di punti per determinare i nove elementi di **E** a meno di una costante moltiplicativa. Per risolvere l'ambiguità data dalla costante moltiplicativa si assume $\|\mathbf{T}\| = \|\mathbf{E}\| = 1$ ottenendo così la matrice essenziale normalizzata.

Anche normalizzando rimane l'ambiguità data dal segno di **E**: infatti se **E** è soluzione del sistema, anche $-\mathbf{E}$ lo è. Ogni matrice essenziale normalizzata (**E** o $-\mathbf{E}$) permette di ottenere due possibili coppie di valori (**R**, **T**), dunque una volta calcolata **E** si possono ricavare quattro possibili soluzioni per la posizione relativa delle due telecamere. Di queste quattro, però, tre possono essere eliminate imponendo che i punti abbiano profondità positiva.

È dunque possibile determinare univocamente la posizione relativa delle due telecamere. La conoscenza di (**R**, **T**) è sufficiente per ricavare la posizione nello spazio del punto ripreso dalle due immagini, mediante il metodo della triangolazione.

Nella discussione precedente si è implicitamente assunto che **E** sia una matrice non nulla, ipotesi necessaria per poter essere normalizzata. Poiché $\mathbf{E} \neq 0$ se e solo se $\mathbf{T} \neq 0$, per applicare la teoria esposta è necessario che i centri ottici delle due telecamere siano distinti. Questo permette di ricavare informazioni sulla scena grazie alla conseguente parallasse.

Data la matrice essenziale con scomposizione ai valori singolari $\mathbf{E} = \mathbf{U}\ \mathrm{diag}(1, 1, 0)\ \mathbf{V}^T$, se la matrice della prima videocamera è $\mathbf{P} = [\mathbf{I} \mid \mathbf{0}]$, ci sono quattro possibili scelte per la matrice **P'** della seconda videocamera, e sono[5]:

$$\mathbf{P'} = [\mathbf{U}\,\mathbf{W}\,\mathbf{V}^T \mid +u3]\ \text{o}\ [\mathbf{U}\,\mathbf{W}\,\mathbf{V}^T \mid -u3]\text{o}\ [\mathbf{U}\,\mathbf{W}^T\,\mathbf{V}^T \mid +u3]\ \text{o}\ [\mathbf{U}\,\mathbf{W}^T\,\mathbf{V}^T \mid -u3] \qquad (14)$$

essendo u3 la terza colonna di **U** e $\mathbf{W} = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$.

Ricordando che la matrice di una videocamera ha la forma $\mathbf{P} = [\mathbf{R} \mid \mathbf{t}]$, vediamo che esistono due possibili risultati sia per R che per t, cosa che dipende dalle possibili configurazioni spaziali delle videocamere, in termini di rotazione e traslazione, che producono punti immagine consistenti con quelli misurati. Si tratta sostanzialmente delle rotazioni o traslazioni delle videocamere che produrrebbero gli stessi punti sull'immagine. Delle quattro configurazioni una sola è quella reale, in cui i punti ricostruiti sono di fronte ad entrambe le videocamere, ed è possibile selezionarla utilizzando un punto dell'immagine come campione, in modo da scegliere la soluzione in cui C − x − X, rispettivamente centro della videocamera, punto immagine e punto ricostruito, e C′ − x′ − X siano allineati in quest'ordine. Ricordiamo che per costruzione $\|u3\| = 1$, ovvero la scala della ricostruzione è fissata una volta imposto come unitario il valore della *baseline*, ossia la distanza tra il centro ottico di fotocamera e proiettore.

È per questo motivo necessario conoscere la distanza tra i due centri ottici per determinare l'orientamento relativo dei due elementi.

Fino ad ora le coordinate del generico punto x_i sono state espresse in coordinate camera $X_i = (\tilde{x}_i\ ,\ \tilde{y}_i\ ,\ \tilde{z}_i)^T \rightarrow x_i = \lambda_i * (\tilde{x}_i / \tilde{z}_i\ ,\ \tilde{y}_i / \tilde{z}_i\ ,\ 1)^T$, se invece si vogliono esprimere i punti in coordinate immagine l'equazione (13) diventa:

$$u_2^T (\mathbf{K_2}^{-1})^T \mathbf{E}\, \mathbf{K_1}^{-1} u_1 = x_2^T \mathbf{F} x_1 = 0 \qquad \text{in cui } \mathbf{F} = (\mathbf{K_2}^{-1})^T \mathbf{E}\, \mathbf{K_1}^{-1} \tag{15}$$

Identificando le proiezioni di otto punti è possibile creare un sistema da cui ricavare **F** e, note le due matrici **K**, si trova la matrice **E**.

Rafforzamento dei vincoli

Come suggerito nel capitolo "Matrice Essenziale e matrice Fondamentale - Determinazione delle matrici" della fonte **[2]**, si può correggere la matrice essenziale, come anche la matrice fondamentale, per fare in modo che faccia parte delle spazio essenziale:

"A causa del rumore, normalmente le matrici ottenute dal sistema lineare non soddisfano il requisito che siano di rango 2 (e nel caso di matrice Essenziale, perciò con un ampio numero di gradi di libertà, che non appartengano proprio al sottospazio delle matrici Essenziali). Una possibile soluzione a questo problema è quella di cercare la matrice più vicina a quella restituita dal sistema lineare che soddisfa però il vincolo sul rango. Tale risultato si ottiene per esempio usando una decomposizione SVD seguita da una composizione:

$$\begin{aligned} \boldsymbol{F} &= \boldsymbol{U}\, diag(r,s,t)\, \boldsymbol{V}^T \\ \boldsymbol{F'} &= \boldsymbol{U}\, diag(r,s,0)\, \boldsymbol{V}^T \end{aligned} \tag{16}$$

Questo procedimento è chiamato constraint enforcement.

La matrice Essenziale rispetto a quella Fondamentale ha in più il vincolo di avere i 2 valori singolari non nulli uguali:

$$\begin{aligned} \boldsymbol{E} &= \boldsymbol{U}\, diag(r,s,t)\, \boldsymbol{V}^T \\ \boldsymbol{E'} &= \boldsymbol{U}\, diag(1,1,0)\, \boldsymbol{V}^T \end{aligned} \tag{17}$$

Se i valori singolari (in seguito a una SVD) della matrice sono 1, la matrice si dice matrice Essenziale normalizzata (normalized essential matrix). La matrice Essenziale ottenuta ponendo D' = diag(1,1,0) è la matrice essenziale normalizzata più vicina a quella data, in accordo con la norma di Frobenius. La matrice Essenziale generata attraverso l'equazione (17) soddisfa la cubic trace-constraint (Demazure, 1988)

$$\boldsymbol{E}\,\boldsymbol{E}^T\,\boldsymbol{E} - \frac{1}{2}\, trace(\boldsymbol{E}\,\boldsymbol{E}^T)\,\boldsymbol{E} = 0 \tag{18}$$

Tale vincolo è condizione necessaria perché la matrice in analisi sia effettivamente Essenziale.

Le matrici ottenute attraverso questo procedimento di rafforzamento soddisfano tutti i requisiti per essere matrici Fondamentali o Essenziali, ma non rappresentano una minimizzazione algebrica, ne tantomeno geometrica, dei vincoli originali."

Algoritmo dei 5 punti

Dato che **E** dipende dalla matrice di rotazione **R** (tre gradi di libertà) e dalla direzione del vettore **T** (due gradi di libertà), ma non dal suo modulo poiché **E** è definita a meno di una costante, la matrice Essenziale è formata da soli 5 gradi di libertà e, in linea teorica, può essere stimata attraverso l'analisi di corrispondenze fra solo 5 punti. L'algoritmo dei 5 punti è di fatto lo standard per la stima della matrice essenziale, tuttavia la richiesta di risolvere un sistema non-lineare rende questo metodo più difficile da implementare rispetto al metodo degli otto punti. Le basi geometriche per il funzionamento di questo tipo di algoritmi, oltre alle varie tecniche che riducono la mole di calcolo necessaria per ricavare la matrice essenziale, sono riportate in **[6].**

Il processo complessivo

Per realizzare una proiezione ortogonale della superficie analizzata che ci permetta di ricavare dati metrici sulle cricche è quindi necessario:

1. Tarare le fotocamere per ottenere la matrice **K** (definita in precedenza con l'espressione (5)) e stimare l'entità dei fenomeni di distorsione;

2. Ricavare la matrice essenziale riconoscendo un minimo di 5 punti all'interno di due fotogrammi;

3. Estrarre dalla matrice essenziale la matrice rotazione **R** ed il vettore traslazione **T** che legano i sistemi di riferimento delle due fotocamere (il modulo di **T** va conosciuto a priori);

4. Note le posizioni delle telecamere, triangolare le coordinate nello spazio di almeno quattro punti presenti sulla superficie;

5. Applicare l'omografia per ottenere dai quattro punti una proiezione ortogonale del piano.

Per determinare la dimensione delle cricche sono quindi sufficienti due immagini in cui siano riconoscibili cinque punti in comune, per migliorare la stima della matrice essenziale si può però usare un numero maggiore di immagini e di punti per ricavare le pose delle fotocamere che minimizzano i *Reprojection error*.

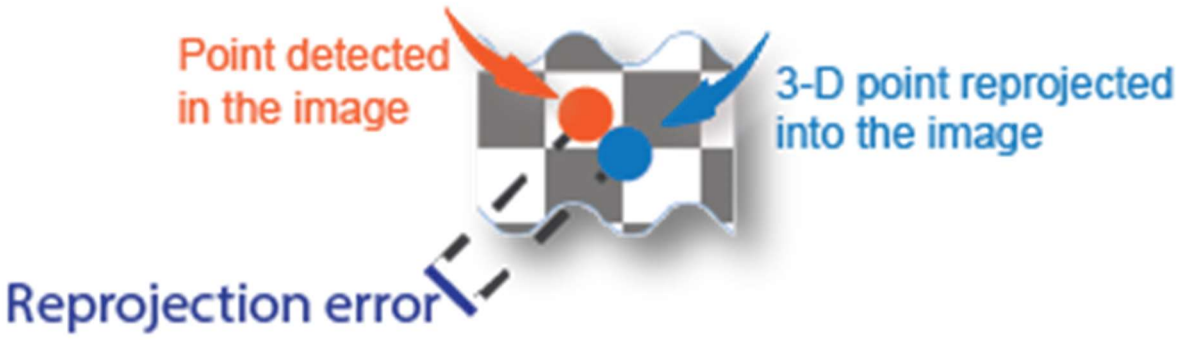

Figura 8: Reprojection error

I *Reprojection error* sono gli scostamenti tra le coordinate di un punto sull'immagine e la proiezione sul piano immagine del punto nello spazio calcolato per triangolazione. Essi nascono da approssimazioni nella stima della matrice essenziale ma soprattutto dal fatto che le coordinate dell'immagine, essendo essa divisa in pixel, presentano un'incertezza.

Per minimizzare questi errori si tenta di lavorare con il maggior numero possibile di fotocamere e proiettori; questo accorgimento può risultare però molto costoso e poco pratico; se invece si utilizza una telecamera in movimento che riprende l'oggetto da una posa differente per ogni fotogramma, si può realizzare la misurazione con un solo strumento. In questo caso, ovviamente, tutti i soggetti che appaiono nella ripresa devono essere fissi nel tempo.

Nel caso si utilizzino più di due immagini è utile ricordare che non è necessario sapere la distanza tra tutte le pose analizzate, ma basta una sola misura: nel caso di una ripresa di una videocamera è sufficiente conoscere la sola distanza percorsa tra la posizione iniziale e quella finale in linea d'aria.

2. Materiali e Metodi

Per creare un programma in grado di triangolare la posizione di un insieme di punti si è utilizzato MatLab poiché MathWorks ha già realizzato e implementato diverse funzioni utili per la monocular visual odometry all'interno degli add-on "Computer Vision Toolbox" e "Image Processing toolbox".

All'interno di questi plug-in troviamo le funzioni:

- `undistortImage`: che compensa la distorsione dell'immagine;
- `detectSURFFeatures` e `extractFeatures`: che estraggono le feature relative ai punti da triangolare per la stima della matrice essenziale;
- `matchFeatures`: che riconosce le stesse feature se presenti in immagini consecutive;
- `estimateEssentialMatrix`: che stima la matrice essenziale con un minimo di 5 punti;
- `relativeCameraPose`: che ricava la posa relativa tra due fotogrammi successivi a partire dalla matrice essenziale;
- `triangulate`: che triangola la posizione nello spazio di una serie di punti usando due fotogrammi successivi;
- `triangulateMultiview`: che triangola le coordinate usando un insieme maggiore di immagini;
- `estimateWorldCameraPose`: che deriva la posa della fotocamera qualora dei punti in essa raffigurati abbiano coordinate mondo note;
- `bundleAdjustment`: che rifinisce le pose e le coordinate mondo dei punti per minimizzare i Reprojection error.

Di queste funzioni solo due non sono deterministiche: estimateWorldCameraPose e bundleAdjustment. Esse non forniscono risultati costanti a parità di input poiché contengono funzioni che generano numeri pseudo-casuali. Per uniformare i risultati, all'interno del programma si riporta il Random Number Generator ad uno stato di default prima di chiamare una di queste due funzioni.

Come si può osservare dai grafici di seguito riportati (**Figura 9**), se non si normalizza il Random Number Generator la traiettoria calcolata può variare nonostante si usi sempre lo stesso video come input.

In questo test si è utilizzato il render reso disponibile dal CVLAB della Tsukuba University[7]. Da tale prova si può anche avere una stima della precisione di questo tipo di algoritmi dato che in colore blu è riportata la traiettoria reale percorsa dalla telecamera (la traiettoria ricavata dal programma è invece rappresentata in verde).

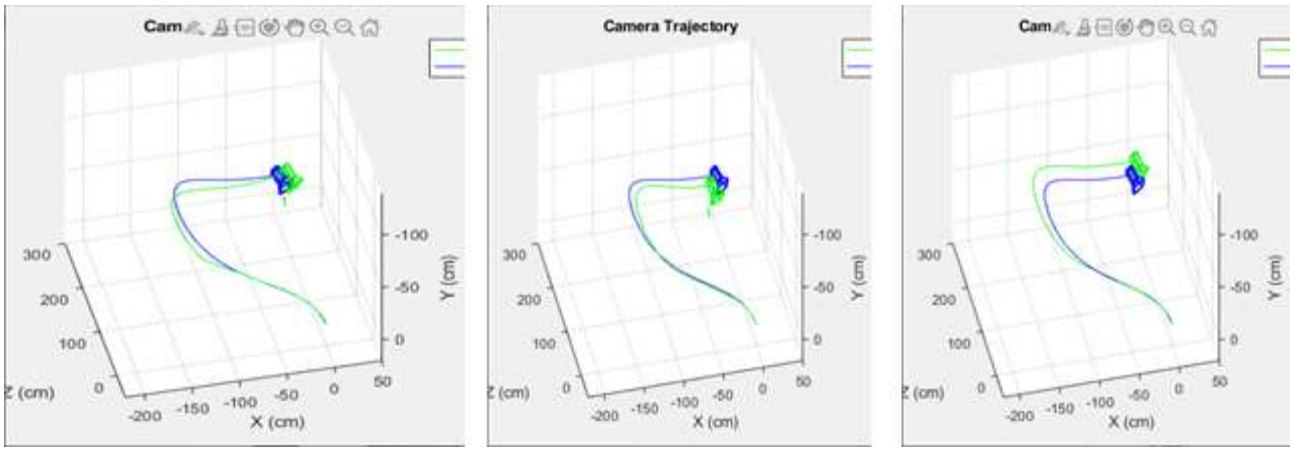

Figura 9: Traiettoria calcolata in tre tentativi utilizzando le immagini della fonte [7]

MatLab offre inoltre uno strumento di calibrazione di telecamere, così da consentire l'individuazione dei parametri intrinseci della camera.

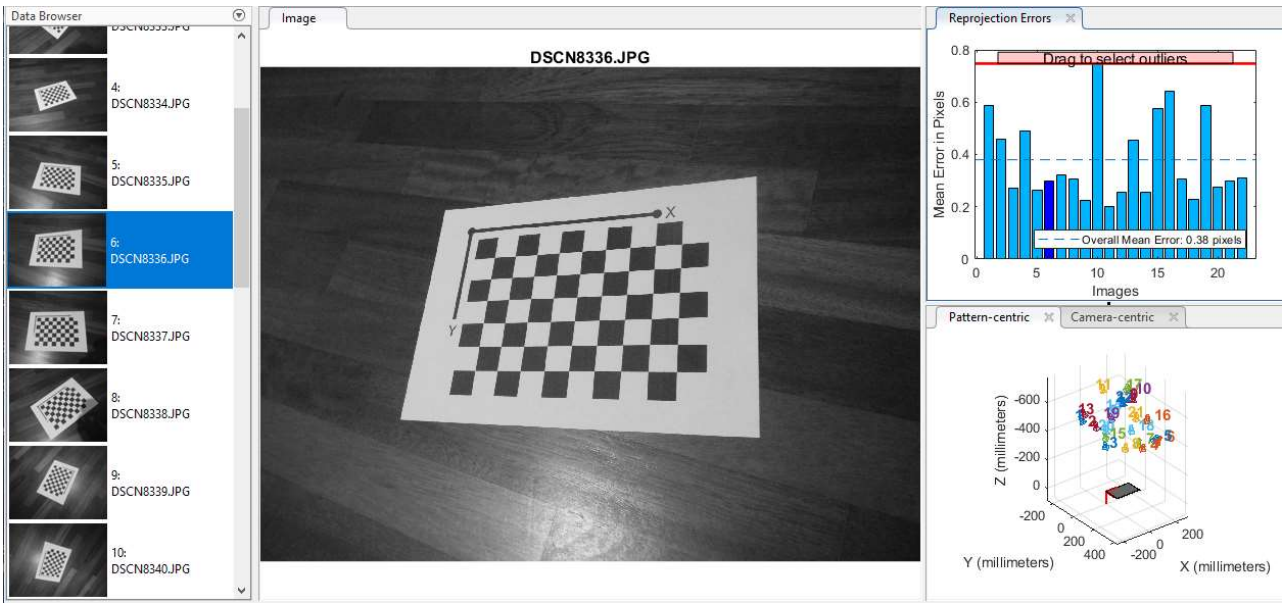

Figura 10: Schermata di calibrazione di una fotocamera in MatLab

Programma per l'estrazione della posa relativa camera-target

Si procede ora a descrivere come questo algoritmo possa derivare la traiettoria di una telecamera e triangolare la posizione di 4 punti colorati (in questo caso il programma cerca il colore verde) a partire da un video acquisito in continuo e dal modulo del vettore spostamento che collega la posa iniziale a quella finale.

Il video deve essere diviso in fotogrammi attraverso un campionamento, queste immagini dovranno poi essere inserite in una cartella chiamata *"odometria"* all'indirizzo specificato nella descrizione.

Il modulo del vettore spostamento va invece inserito modificando il valore assegnato a "`distanceCovered`" nelle prime righe del programma.

```
%Pulizia Workspace e Command Window
clear; clc;

%Caricamento immagini nel workspace e creazione di un "imageviewset" vuoto
images = imageDatastore(fullfile(toolboxdir('vision'), 'visiondata', ...
      'odometria'));
vSet = imageviewset;

%Definizione della distanza percorsa dalla fotocamera in linea d'aria
distanceCovered=100;
```

In questa sezione, oltre ad operazioni iniziali come il rimuovere elementi precedenti dal Workspace e Command Window, si caricano all'interno del programma i fotogrammi di input. Il programma cerca le immagini all'interno della cartella in cui è installata la toolbox "vision", per ricavare dove sia posizionata tale cartella è sufficiente inserire `toolboxdir('vision')` nella Command Window e premere invio. In tale indirizzo, nella sottocartella "visiondata", è necessario creare una nuova cartella sotto il nome di "odometria" in cui inserire il campionamento del video.

In questo passaggio si fornisce anche il secondo input del programma: *distanceCovered*. Questo valore deve essere modificato dall'utente in modo da farlo corrispondere alla distanza tra le posizioni che il punto principale della fotocamera assume quando cattura il fotogramma iniziale e quello finale.

Questo secondo input è necessario perché la tecnica odometrica, come già discusso in precedenza, non è in grado di ricavare il modulo del vettore spostamento della telecamera che collega due fotogrammi successivi, ma solo le altre due coordinate polari. È quindi necessario fornire al programma almeno una misura reale in modo che, dopo aver calcolato tutti gli spostamenti applicando una scala di default, esso possa convertire tutti gli spostamenti alla dimensione corretta al termine del calcolo della traiettoria.

```
%Lettura del primo fotogramma
viewId = 1;
```

```
Irgb = readimage(images, 1);

%Definizione di matrici utili al programma
CPoints = zeros(4, 2, numel(images.Files));
CMatches = zeros(4, 2, numel(images.Files) - 1);
blacklist = zeros(1,numel(images.Files));
blacklist_i = 1;
```

In questa sezione vengono inizializzate delle variabili (double o matrici di double) contenenti informazioni necessarie al funzionamento dell'algoritmo, oltre alla lettura da parte del programma del primo fotogramma per mezzo della variabile *Irgb*.

viewID indica quale fotogramma sta venendo processato in un dato momento.

CPoints è una matrice 4x2x*m* in cui *m* è in numero di immagini fornite al programma: ogni riga serve per salvare le coordinate immagine (x, y) di uno dei quattro punti proiettati per ogni fotogramma. Il numero assegnato alla terza dimensione indica a quale fotogramma si riferiscono i punti, per esempio la tabella 4x2 alla profondità *n* contiene le coordinate dei quattro punti rappresentati nell'*n*-esimo fotogramma.

CMatches è una matrice 4x2x(*m*-1) in cui *m* è in numero di immagini fornite al programma: dato che all'interno di *CPoints* ogni immagine ha le coordinate dei propri punti salvate in ordini differenti, la matrice *CMatches* contiene gli accoppiamenti che la funzione "matchFeatures" realizza tra i punti di un fotogramma e quello successivo. La matrice può essere pensata come una successione di matrici 4x2: la colonna di sinistra rappresenta i punti della prima immagine (in cui 1 è il punto descritto dalla prima riga di *CPoints*, 2 è la seconda e così via) mentre la seconda colonna rappresenta i punti del secondo fotogramma, se due punti si trovano nella stessa riga di *CMatches* vuol dire che il punto a sinistra nella prima immagine corrisponde a quello a destra nella seconda. Il numero assegnato nella terza dimensione rappresenta gli accoppiamenti tra il fotogramma corrispondente a tale numero e quello successivo; qualora l'n-esimo fotogramma non sia considerato per il calcolo della posizione dei punti (ossia se fa parte della *blacklist* di cui parleremo a breve) allora la matrice *CMatches(:,:,n-1)*, che avrebbe dovuto contenere i collegamenti tra n-1 e n, conterrà solo zeri mentre in *CMatches(:,:,n)* saranno salvati gli accoppiamenti di punti tra n-1 e n+1.

blacklist è una matrice riga che contiene tutti i *viewID* dei fotogrammi che non sono utilizzati per la stima della posizione dei quattro punti proiettati, ciò può avvenire perché MatLab non è in grado di riconoscere tutti e quattro i punti o perché sono stati trovati meno di tre accoppiamenti tra i punti di quel fotogramma e quelli dell'ultima immagine che non fa parte della *blacklist*.

blacklist_i rappresenta la posizione in cui deve essere posto il prossimo *viewID* che deve far parte della *blacklist*, esso è anche utile perché sottraendovi 1 si ottiene il numero totale di immagini in *blacklist*.

```
%Definizione dei parametri della telecamera
focalLength = [560 560]; % specified in units of pixels
imageSize = size(Irgb,[1,2]); % in pixels [mrows, ncols]
principalPoint = [imageSize(1,2)./2 imageSize(1,1)./2]; % in pixels [x, y]
```

```
intrinsics = cameraIntrinsics(focalLength, principalPoint, imageSize);

%Correzione della distorsione
prevI = undistortImage(rgb2gray(Irgb), intrinsics);
Irgb = undistortImage(Irgb, intrinsics);
```

In questa sezione si definiscono le caratteristiche della fotocamera utilizzata e vengono riassunte nell'oggetto *intrinsics*, tramite esso è possibile creare due versioni del primo fotogramma del video: *prevI*, in cui si compensa la distorsione della lente e si converte in bianco e nero, e *Irgb*, aggiornata con solo la prima operazione. La prima verrà utilizzata per comprendere la traiettoria realizzata dalla telecamera mentre la seconda sarà sfruttata per identificare i punti colorati.

Si può notare come la distanza focale qui indicata sia misurata in pixel e attraverso due numeri reali, questi valori indicano rispettivamente i prodotti $F \times sx$ e $F \times sy$ dove F è la distanza focale reale (misurata per esempio in millimetri). *sx* e *sy* sono invece i fattori di conversione pixel/unità di lunghezza (pixel/mm in questo caso) applicato dal sensore della fotocamera rispettivamente sull'asse orizzontale e verticale.

Modificando questi valori per farli coincidere con quelli della telecamera usata è quindi necessario ricordare che cambiando la risoluzione delle immagini catturate si deve correggere il parametro *focalLength* di conseguenza.

```
%Rilevazione delle feature per l'odometria
prevPoints = detectSURFFeatures(prevI, 'MetricThreshold', 500);
numPoints = 200;
prevPoints = selectUniform(prevPoints, numPoints, size(prevI));
prevFeatures = extractFeatures(prevI, prevPoints, 'Upright', true);

%aggiunta dei punti necessari all'odometria e salvataggio della posizione
%iniziale
vSet = addView(vSet, viewId, rigid3d(eye(3), [0 0 0]), 'Points', prevPoints);

camPose = poses(vSet);
```

In questa sezione si estraggono 200 punti dalla prima immagine in bianco e nero per poi creare delle feature da confrontare con quelle dei fotogrammi successivi. Questi punti vengono poi inseriti all'interno dell'*imageviewset* e la posa iniziale viene salvata per essere poi utilizzata nel plot della traiettoria.

L'opzione "upright" viene utilizzata perché migliora il riconoscimento delle feature nel caso in cui la telecamera non abbia rotazioni consistenti nel piano immagine; ove invece l'applicazione richieda questo tipo di movimento basta eliminare "`'Upright', true`" dalla funzione "extractFeatures".

```
%Generazione di un SURFPoint per ogni punto del colore scelto
[prevCPoints, blacklist, blacklist_i] = helperFindCPoints(Irgb, ...
    blacklist, blacklist_i, viewId);
```

```
%Se sono stati identificati tutti i punti, avviene l'estrazione delle
%features corrispondenti e le posizioni di tali punti vengono salvate
%nell'apposita tabella
if blacklist_i == 1
    prevCFeatures = extractFeatures(prevI, prevCPoints, ...
        'FeatureSize', 128, 'Upright', true);
    CPoints(:,:,1) = prevCPoints.Location;
end
```

In questa sezione si utilizza il sottoprogramma "helperFindCPoints" (riportato in seguito) per identificare i punti proiettati e ottenerne dei SURFpoints sotto il nome *prevCPoints*. *blacklist* e *blacklist_i* vengono posti anche tra gli output, oltre che tra gli input, per aggiornarli in modo da essere utilizzati per i fotogrammi successivi.

Se vengono identificati tutti e quattro i punti allora si estraggono le feature da confrontare con i fotogrammi successivi e le coordinate ottenute vengono salvate nel primo livello di *CPoints*.

HelperFindCPoints

```
function [currSURFPoints, blacklist, blacklist_i] = ...
    helperFindCPoints(Irgb, blacklist, blacklist_i, viewId)

    %conversione da rgb a hsv
    Ihsv  = rgb2hsv(Irgb);
```

Come prima cosa si converte l'immagine (la cui distorsione era già stata compensata in precedenza) da rgb a hsv.

Ihsv è quindi costituita da 3 matrici delle dimensioni dell'immagine (in pixel): la prima (il valore h) indica la tonalità (Hue) di ogni punto, la seconda (il valore s) la saturazione (Saturation), mentre l'ultima (il valore v) è la luminosità (Value).

Irgb rimane invece costituita da tre matrici che indicano i valori di rosso, verde e blu di ogni punto.

```
    %Creazione della maschera
    h = Ihsv(:,:,1) >= 0.28) & (Ihsv(:,:,1) <= 0.38);
    s = (Ihsv(:,:,2) >= 0.4) & (Ihsv(:,:,2) <= 1);
    v = (Ihsv(:,:,3) >= 0) & (Ihsv(:,:,3) <= 1);
    mask = uint8(h & s & v);

    %Esclusione dei punti con non rispettano i parametri della maschera
    mR = mask .* Irgb(:,:,1);
    mG = mask .* Irgb(:,:,2);
    mB = mask .* Irgb(:,:,3);
    Icolor = cat(3, mR, mG, mB);
```

In questo caso si vogliono cercare quattro punti di colore verde: dato che il verde puro ha una tonalità di 1/3 si filtra l'immagine con una maschera, ossia una matrice bidimensionale delle stesse dimensioni di *Ihsv* (e quindi di *Irgb*) che ha tutti i valori pari a zero, tranne per i pixel dove l'immagine ha valori di h compresi in [0.28,0.38] e saturazione maggiore del 40% a cui corrisponde il valore 1.

Applicando questa maschera ai tre livelli di *Irgb* e ri-assemblandoli si ottiene una versione del fotogramma in cui sono presenti solo i pixel di colore verde.

```
%Convesione da rgb a binario
Igray = rgb2gray(Icolor);
Ibw = imbinarize(Igray,graythresh(Igray));
Ilabel = logical(Ibw);

%Identificazione delle figure colorate e selezione delle quattro con
%area maggiore
stat = regionprops(Ilabel,'boundingbox','Area');
[~,I] = maxk([stat.Area],4);
```

L'immagine così ottenuta viene ridotta ad una sola matrice booleana dove il valore 1 è solo presente in corrispondenza dei punti colorati di verde. Da essa, con la funzione "regionprops", è possibile estrarre informazioni sulle regioni di color verde presenti nell'immagine: "area" misura semplicemente l'area di ogni regione mentre "boundingbox" genera un rettangolo (orientato lungo gli assi x e y) tale da racchiudere ogni regione, salvandone l'origine e le dimensioni.

In questo modo possiamo ricavare le quattro regioni con area maggiore in modo da escludere eventuali punti che possono essere considerati verdi e lasciare solo i punti desiderati.

```
%Creazione di un SURFPoint per ognuna delle 4 figure
currSURFPoints = SURFPoints;
k = 0;
for i=1:4
    mask_i = uint8(createMask ...
        (images.roi.Rectangle('Position',stat(I(i)).BoundingBox), Icolor));

    mR = mask_i .* Icolor(:,:,1);
    mG = mask_i .* Icolor(:,:,2);
    mB = mask_i .* Icolor(:,:,3);
    Icolor_i = cat(3, mR, mG, mB);

    currPoint = detectSURFFeatures(rgb2gray(Icolor_i));
```

Qui inizia un ciclo che prosegue fino all'"end" finale della prossima sezione: esso permette di usare i "boundingbox" creati in precedenza come delle maschere per isolare le parti della foto dove si trovano i singoli punti; da ognuna di queste regioni si estraggono i SURFpoint.

```
        if size(currPoint.Location,1) > 0
            currPoint = currPoint.selectStrongest(1);
            currSURFPoints(i) = currPoint;
            k = k+1;
        end

    end
```

Qualora nell'area considerata si rilevino più punti si considera solo quello principale e lo si pone all'interno dell'oggetto *currSURFPoints* nella posizione i-esima.

Nel caso in cui non si sia potuto ricavare alcun punto dalla regione considerata, la variabile *k* non potrà essere aumentata in quell'iterazione e dunque, quando il ciclo verrà riprodotto la quarta ed ultima volta, il valore finale di *k* non potrà raggiungere il numero 4.

```
    %Se non vengono trovati tutti i punti il fotogramma viene posto nella
    %blacklist
    if k ~= 4
        blacklist(blacklist_i) = viewId;
        blacklist_i = blacklist_i+1;
    end
```

Nel caso almeno un ciclo non produca un SURFpoint (ossia nel caso in cui *k* sia minore di 4) il fotogramma attuale viene posto nella *blacklist* facendo aumentare *blacklist_i* di un'unità.

```
%Lettura e compensazione della distorsione per il secondo fotogramma
viewId = 2;
Irgb = readimage(images, viewId);
I = undistortImage(rgb2gray(Irgb), intrinsics);
Irgb = undistortImage(Irgb, intrinsics);

%Creazione dei SURFPoint e delle features utili all'odometria nella seconda
%immagine. Accoppiamento delle features tra la prima e la seconda immagine
[currPoints, currFeatures, indexPairs] = helperDetectAndMatchFeatures(...
    prevFeatures, I);
```

Si passa al secondo fotogramma e dopo aver caricato e corretto la distorsione dell'immagine si procede col sottoprogramma "helperDetectAndMatchFeatures": esso è un programma realizzato dalla MathWorks e raggruppa le operazioni di estrazione dei SURFPoints, delle feature e degli accoppiamenti con le feature del fotogramma precedente.

Si riporta qui il testo di tale sottoprogramma:

```
% helperDetectAndMatchFeatures Detect, extract, and match features
%   [currPoints, currFeautres, indexPairs] = helperDetectAndMatchFeatures(
%   prevFeatures, I) detects and extract features from image I and matches
%   them to prevFeatures. prevFeatures is an M-by-N matrix fof SURF
%   descriptors. I is a grayscale image. currPoints are the SURF points
%   detected in image I, and currFeatures are the corresponding SURF
%   descriptors. indexPairs is an M-by-2 matrix containing the indices of
%   matches between prevFeatures and currFeatures.
%
%   See also detectSURFFeatures, selectUniform, extractFeatures,
%   matchFeatures

% Copyright 2016 The MathWorks, Inc.

function [currPoints, currFeatures, indexPairs] = ...
      helperDetectAndMatchFeatures(prevFeatures, I)

numPoints = 150;

% Detect and extract features from the current image.
currPoints   = detectSURFFeatures(I, 'MetricThreshold', 500);
currPoints   = selectUniform(currPoints, numPoints, size(I));
currFeatures = extractFeatures(I, currPoints, 'Upright', true);

% Match features between the previous and current image.
indexPairs = matchFeatures(prevFeatures, currFeatures, 'Unique', true);
```

```
%Stima della posa della seconda immagine
[orient, loc, inlierIdx] = helperEstimateRelativePose(...
    prevPoints(indexPairs(:,1)), currPoints(indexPairs(:,2)), intrinsics);

%Filtraggio dei punti usati per l'odometria
indexPairs = indexPairs(inlierIdx, :);

%Aggiunta dei punti e degli accoppiamenti all'"imageviewset"
vSet = addView(vSet, viewId, rigid3d(orient, loc), 'Points', currPoints);
vSet = addConnection(vSet, viewId-1, viewId, 'Matches', indexPairs);
```

Si usa poi un altro sottoprogramma della MathWorks chiamato “helperEstimateRelativePose” per applicare le tecniche di monocular visual odometry e ricavare l’orientamento e la posizione relativa tra la prima e la seconda fotocamera sfruttando 5 o più punti.

Dato che la posa assoluta della prima telecamera è stata posta come orientata verso l’asse z e coincidente con l’origine, la posa relativa della seconda telecamera coincide con la sua posa assoluta.

Si può notare come *prevPoints* e *currPoints*, quando usati come input di “helperEstimateRelativePose”, vengono ordinati secondo gli accoppiamenti “indexPairs”, determinati

in precedenza da "matchFeatures". Ciò permette di escludere i punti che non vengono riconosciuti in entrambe le immagini oltre a consentire di ordinare l'elenco di punti in modo che ad ogni riga corrisponda lo stesso punto reale sia in *prevPoints* che in *currPoints*.

Si riporta qui il testo di tale sottoprogramma:

```
% helperEstimateRelativePose Robustly estimate relative camera pose
%  [orientation, location, inlierIdx] = helperEstimateRelativePose(
%    matchedPoints1, matchedPoints2, cameraParams) returns the pose of
%  camera 2 in camera 1's coordinate system. The function calls
%  estimateEssentialMatrix and cameraPose functions in a loop, until
%  a reliable camera pose is obtained.
%
%  Inputs:
%  -------
%  matchedPoints1 - points from image 1 specified as an M-by-2 matrix of
%                   [x,y] coordinates, or as any of the point feature types
%  matchedPoints2 - points from image 2 specified as an M-by-2 matrix of
%                   [x,y] coordinates, or as any of the point feature types
%  cameraParams   - cameraParameters object
%
%  Outputs:
%  --------
%  orientation - the orientation of camera 2 relative to camera 1
%                specified as a 3-by-3 rotation matrix
%  location    - the location of camera 2 in camera 1's coordinate system
%                specified as a 3-element vector
%  inlierIdx   - the indices of the inlier points from estimating the
%                fundamental matrix
%
%  See also estimateEssentialmatrix, estimateFundamentalMatrix,
%  relativeCameraPose

% Copyright 2016 The MathWorks, Inc.

function [orientation, location, inlierIdx] = ...
    helperEstimateRelativePose(matchedPoints1, matchedPoints2, cameraParams)

if ~isnumeric(matchedPoints1)
    matchedPoints1 = matchedPoints1.Location;
end

if ~isnumeric(matchedPoints2)
    matchedPoints2 = matchedPoints2.Location;
end

for i = 1:100
    % Estimate the essential matrix.
    [E, inlierIdx] = estimateEssentialMatrix(matchedPoints1, matchedPoints2,...
        cameraParams);

    % Make sure we get enough inliers
    if sum(inlierIdx) / numel(inlierIdx) < .3
        continue;
    end
```

```
    % Get the epipolar inliers.
    inlierPoints1 = matchedPoints1(inlierIdx, :);
    inlierPoints2 = matchedPoints2(inlierIdx, :);

    % Compute the camera pose from the fundamental matrix. Use half of the
    % points to reduce computation.
    [orientation, location, validPointFraction] = ...
        relativeCameraPose(E, cameraParams, inlierPoints1(1:2:end, :),...
        inlierPoints2(1:2:end, :));

    % validPointFraction is the fraction of inlier points that project in
    % front of both cameras. If the this fraction is too small, then the
    % fundamental matrix is likely to be incorrect.
    if validPointFraction > .8
       return;
    end
end

% After 100 attempts validPointFraction is still too low.
error('Unable to compute the Essential matrix');
```

```
%Identificazione delle posizioni e delle features dei punti proiettati.
%Accoppiamento delle features tra la prima e la seconda immagine
[CPoints(:,:,viewId), currCFeatures, currCMatches, blacklist, blacklist_i] = ...
    helperDetectAndMatchFeaturesC(prevCFeatures, Irgb, blacklist, ...
        blacklist_i, viewId);

%Nel caso si trovino solo 3 accoppiamenti tra i punti delle due immagini si
%collegano tra loro i punti non accoppiati
CMatches(1:size(currCMatches,1),:,1) = currCMatches;
if size(currCMatches,1) == 3
    CMatches(4,:,1) = [setdiff(1:4,CMatches(:, 1, 1)) ...
        setdiff(1:4,CMatches(:, 2, 1))];
end
```

In questa sezione si impiega il sottoprogramma "helperDetectAndMatchFeaturesC" in modo simile a come si è utilizzato "helperDetectAndMatchFeatures" nelle sezioni precedenti, solo che in questo caso ad essere estratti, convertiti in feature e accoppiati con l'immagine precedente sono i punti proiettati.

HelperDetectAndMatchFeaturesC

```
function [currPoints, currFeatures, indexPairs,  blacklist, blacklist_i] =...
    helperDetectAndMatchFeaturesC(prevFeatures, Irgb, blacklist, ...
        blacklist_i, viewId)

%Generazione di un SURFPoint per ogni punto del colore scelto
[currSURFPoints,  blacklist, blacklist_i] = ...
```

```
    helperFindCPoints(Irgb, blacklist, blacklist_i, viewId);
```

Come visto in precedenza per il primo fotogramma, il sottoprogramma "helperFindCPoints" estrae i quattro punti colorati sotto forma di SURFpoints o pone il fotogramma corrente nella *blacklist*.

```
if ismember(viewId ,blacklist)
    %Valori di default da riportare nel caso non vengano riconosciuti i
    %punti
    currPoints = zeros(4,2);
    currFeatures = [];
    indexPairs = zeros(4,2);
else
    %Estrazione delle features e salvataggio delle posizioni dei punti
    I = rgb2gray(Irgb);
    currFeatures = extractFeatures(I, currSURFPoints, ...
        'FeatureSize', 128, 'Upright', true);
    currPoints = currSURFPoints.Location;
```

Se "helperFindCPoints" non ha trovato tutti e quattro i punti "helperDetectAndMatchFeaturesC" riporta come output i valori all'interno dell'"if", in caso contrario si procede a convertire l'immagine in bianco e nero perché la funzione "extractFeatures" qui utilizzata accetta solo quel tipo di input.

Le coordinate dei punti vengono salvate a parte all'interno della matrice *currPoints* (si ricorda che *currSURFPoints* fa invece parte della classe SURFPoints).

```
    %Interruzione del sottoprogramma se non ci sono fotogrammi precedenti
    %adatti ad essere confrontati con il fotogramma studiato
    if all(ismember(1:viewId-1, blacklist))
        indexPairs = zeros(4,2);
        return
    end

    %Riconoscimento dei punti proiettati tra il fotogramma attuale e
    %l'ultimo fotogramma utile
    indexPairs = matchFeatures(prevFeatures, currFeatures, ...
        'MatchThreshold', 100, 'MaxRatio', 1, 'Unique', true);
```

Nel caso in cui nel fotogramma corrente si siano rilevati tutti e quattro i punti ma non si abbiano immagini con cui confrontarlo perché quest'ultime fanno tutte parte della *blacklist*, allora il sottoprogramma si arresta e fornisce al programma principale solo le coordinate dei punti e le feature rilevate.

Nel caso in cui si trovi un fotogramma utilizzabile, si usa "matchFeatures" tra le feature correnti e quelle dell'ultima immagine disponibile.

```
    %Se ci sono almeno 3 punti viene fornita al programma la matrice
    %indexPairs, in caso contrario si restituisce il valore di default ed
    %il fotogramma corrente viene posto nella blacklist
    if size(indexPairs, 1) > 2
         return
    end

    indexPairs = zeros(4,2);
    blacklist(blacklist_i) = viewId;
    blacklist_i = blacklist_i+1;

end
```

Se "matchFeatures" rileva meno di tre accoppiamenti il risultato non viene fornito al programma principale (si riporta invece la matrice di default costituita da soli zeri) e il fotogramma viene posto nella blacklist.

Inoltre, una volta ottenuti gli accoppiamenti dei punti tra la prima e la seconda immagine che vengono provvisoriamente salvati in *currCMatches*, si cerca di correggere i risultati ottenuti nel caso in cui scaturiscano solo tre accoppiamenti. Nel caso non si trovi somiglianza tra due feature nelle due immagini, ma tutti e tre gli altri punti sono stati riconosciuti a dovere, si può dedurre che i punti con cui il matching non ha avuto successo sono quelli rimasti non accoppiati nei due fotogrammi.

Questa tecnica ci permette di ridurre drasticamente i fotogrammi che vanno a far parte della *blacklist*, nonostante aumenti il rischio che il programma confonda due punti tra loro.

```
%Preparazione per il fotogramma successivo
prevI = I;
prevFeatures = currFeatures;
prevPoints = currPoints;
if ~ismember(2,blacklist)
    prevCFeatures = currCFeatures;
end
```

Queste sono le ultime operazioni con cui si lavora con la seconda immagine, dal fotogramma successivo le funzioni saranno applicate all'interno di un ciclo "for" e quindi si fanno equivalere le variabili "prev" (che si riferiscono al ciclo precedente) a quelle del secondo fotogramma.

Una distinzione va fatta per *prevCFeatures* poiché, se la seconda immagine fa parte della *blacklist* e la prima no, il fotogramma successivo dovrà confrontare i suoi punti colorati con la prima immagine e non con la seconda; non si dovrà quindi aggiornare questa variabile nel caso in cui 2 sia un elemento della *blacklist*.

```
%Valori necessari a suddividere il campione di fotogrammi in funzione del
```

```
%loro numero
windowSize = double(idivide(numel(images.Files),int16(5)));
windowSize2 = double(idivide(numel(images.Files),int16(10)));

%Definizione di un flag necessario alla creazione della whitelist
jDetermined = false;
```

Si trovano qui alcuni valori da inizializzare prima di avviare il ciclo "for".

windowSize (definito come 1/5 del numero totale di fotogrammi, approssimato per difetto) indica l'ultimo fotogramma che otterrà una stima migliore della propria posa.

Le pose dei fotogrammi successivi a questo verranno calcolate utilizzando solo una porzione delle immagini precedenti: il numero dei fotogrammi utilizzati è definito da *windowSize2* (ossia 1/10 del numero totale di fotogrammi, approssimato per difetto).

jDetermined è invece una variabile booleana che passerà al valore 1 quando il numero *j* verrà calcolato. Come si vedrà in seguito *j* è il *viewID* del primo fotogramma che non fa parte della *blacklist* e, dato che non può essere modificato dopo il suo calcolo iniziale, è necessario trovarlo una sola volta.

```
for viewId = 3:windowSize
    %Lettura immagine
    Irgb = readimage(images, viewId);

    %Compensazione della distorsione
    I = undistortImage(rgb2gray(Irgb), intrinsics);
    Irgb = undistortImage(Irgb, intrinsics);
```

In questa sezione inizia il primo dei due cicli "for" che realizzano tutte le operazioni viste in precedenza dal terzo all'ultimo fotogramma. Il primo ciclo, che arriva fino al valore *windowSize*, tiene in considerazione tutti le immagini precedenti a quella attuale per stimare la posa della videocamera; come si vedrà in seguito il secondo ciclo terrà conto di un numero minore di immagini.

Le prime operazioni del ciclo sono costituite dalla lettura e dalla compensazione della distorsione per l'immagine in bianco e nero e per quella a colori.

```
    %Creazione dei SURFPoint e delle features utili all'odometria nell'immagine
    %corrente. Accoppiamento delle features tra la prima e la seconda immagine
    [currPoints, currFeatures, indexPairs] = helperDetectAndMatchFeatures( ...
        prevFeatures, I);

    %Filtraggio dei punti usati per l'odometria
    inlierIdx = helperFindEpipolarInliers(prevPoints(indexPairs(:,1)), ...
        currPoints(indexPairs(:, 2)), intrinsics);
    indexPairs = indexPairs(inlierIdx, :);
```

Si è già visto come "helperDetectAndMatchFeatures" fornisca per il fotogramma corrente SURFPoints, feature e accoppiamenti con l'immagine precedente ma, al contrario della seconda foto, si utilizza "helperFindEpipolarInliers" al posto di "helperEstimateRelativePose". L'unica differenza tra i due sottoprogrammi è che il primo non fornisce come output orientamento e posizione della fotocamera, poiché essi verranno stimati in modo differente. L'unico output fornito è quindi l'elenco degli inliner il quale viene usato per escludere tutti gli accoppiamenti che si riferiscono agli outliners.

Si riporta qui il testo del sottoprogramma nella sua semplicità:

```
% helperFindEpipolarInliers Find epipolar inliers among matched image points
%  inlierIdx = helperFindEpipolarInliers(matchedPoints1, matchedPoints2,
cameraParams)
%  returns indices of matched image points which satisfy the epipolar
%  constraint.
%
%  matchedPoints1 and matchedPoints2 are sets of matched image points
%  specified as M-by-2 matrices of [x,y] coordinates, or as any of the point
%  feature types. cameraParams is a cameraParameters object. inlierIdx is a
%  logical index of the epipolar inliers.
%
%  See also estimateEssentialMatrix, estimateFundamentalMatrix,
%  relativeCameraPose

% Copyright 2016 The MathWorks, Inc.

function inlierIdx = helperFindEpipolarInliers(matchedPoints1, ...
    matchedPoints2, cameraParams)

% Use the inlierIdx output from helperEstimateRelativePose and ignore the
% orientation and rotation.
[~, ~, inlierIdx] = helperEstimateRelativePose(matchedPoints1, ...
    matchedPoints2, cameraParams);
```

```
    %Triangolazione dei punti trovati nell'immagine corrente usando i due
    %fotogrammi precedenti per determinare le loro coordinate mondo e
    %coordinate immagine (nel fotogramma attuale)
    [worldPoints, imagePoints] = helperFind3Dto2DCorrespondences(vSet, ...
        intrinsics, indexPairs, currPoints);
```

In questa sezione troviamo un altro sottoprogramma realizzato dalla MathWorks per triangolare la posizione dei punti visibili nell'immagine corrente sfruttando le loro coordinate immagine nei due fotogrammi precedenti.

Il programma inoltre filtra i punti dell'immagine corrente in modo da escludere quelli che non trovano corrispondenza nei due fotogrammi precedenti.

Si riporta qui il testo del sottoprogramma:

```
% helperTriangulateLastFrames triangulate points from last two views
%  [worldPoints, imagePoints] = helperTriangulateLastFrames(vSet,
%  cameraParams, matchIdx, currPoints) returns world points triangulated
%  from the last two views in the view set, which are also visible in the
%  current image.
%
%  vSet is a viewSet object, cameraParams is a cameraParameters object, and
%  matchIdx is an M-by-2 matrix of matched indices between the points in
%  the last view of vSet and the current image.
%
%  worldPoints is a three-column matrix containing the [x,y,z] coordinates
%  of world points which are visible in the last two views of vSet and the
%  current image. idxTriplet is a vector containing the indices of points
%  in the current image corresponding to worldPoints.
%
%  See also triangulate, imageviewset, estimateWorldCameraPose

% Copyright 2016-2019 The MathWorks, Inc.
function [worldPoints, imagePoints] = helperFind3Dto2DCorrespondences(vSet, ...
    cameraParams, matchIdx, currPoints)

camPoses = poses(vSet);

% Compute the camera projection matrix for the next-to-the-last view.
loc1 = camPoses.AbsolutePose(end-1).Translation;
orient1 = camPoses.AbsolutePose(end-1).Rotation;
[R1, t1] = cameraPoseToExtrinsics(orient1, loc1);
camMatrix1 = cameraMatrix(cameraParams, R1, t1);

% Compute the camera projection matrix for the last view.
loc2 = camPoses.AbsolutePose(end).Translation;
orient2 = camPoses.AbsolutePose(end).Rotation;
[R2, t2] = cameraPoseToExtrinsics(orient2, loc2);
camMatrix2 = cameraMatrix(cameraParams, R2, t2);

% Find indices of points visible in all three views.
matchIdxPrev = vSet.Connections.Matches{end};
[~, ia, ib] = intersect(matchIdxPrev(:, 2), matchIdx(:, 1));
idx1 = matchIdxPrev(ia, 1);
idx2 = matchIdxPrev(ia, 2);
idxTriplet = matchIdx(ib, 2);

% Triangulate the points.
points1 = vSet.Views.Points{end-1};
points2 = vSet.Views.Points{end};
worldPoints = triangulate(points1(idx1,:), ...
    points2(idx2,:), camMatrix1, camMatrix2);
imagePoints = currPoints(idxTriplet).Location;
```

```
    %Si fa in modo che RANSAC possa realizzare un numero di cicli maggiore
```

```
    %rispetto al massimo fissato
    warningstate = warning('off','vision:ransac:maxTrialsReached');

    %Estrazione di orientamento e posizione della posizione della
    %telecamera nell'istante attuale
    rng default;
    [orient, loc] = estimateWorldCameraPose(imagePoints, worldPoints, ...
        intrinsics);

    %Ripristino dello stato di default
    warning(warningstate)

    %Aggiunta dei punti e degli accoppiamenti all'"imageviewset"
    vSet = addView(vSet, viewId, rigid3d(orient, loc), 'Points', currPoints);
    vSet = addConnection(vSet, viewId-1, viewId, 'Matches', indexPairs);
```

Con "estimateWorldCameraPose" si stima la posa del fotogramma corrente utilizzando gli output della funzione precedente. Si è scelto di utilizzare questo metodo al posto della funzione "helperEstimateRelativePose", usata per la posa del secondo fotogramma, perché quest'ultima lavora con l'ipotesi che tra le due immagini (in questo caso la seconda e la terza) ci sia stato uno spostamento unitario; applicare quindi questa ipotesi sia alla transizione tra primo e secondo fotogramma che allo spostamento tra il secondo e il terzo sarebbe lecito solo se la telecamera si muovesse sempre a velocità costante.

Triangolando prima le posizioni dei punti nello spazio si riesce invece a trovare la posizione e l'orientamento corretti (a meno di un fattore di scala che si compenserà in seguito).

"estimateWorldCameraPose" è una delle due funzioni all'interno del programma che non fornisce un risultato deterministico, per uniformare i risultati la linea precedente alla funzione pone il Random Number Generator allo stato iniziale.

È inoltre consigliato di far ignorare il warning del massimo numero di cicli al programma RANSAC quando viene sfruttato da "estimateWorldCameraPose" perché i risultati ottenuti sono comunque validi nonostante il livello di confidenza non raggiunga quello impostato nella funzione (la linea successiva a "estimateWorldCameraPose" annulla tale modifica).

```
    %Rifinitura delle posizioni ricavate utilizzando tutti i fotogrammi a
    %partire dal primo
    tracks = findTracks(vSet);
    camPoses = poses(vSet);
    xyzPoints = triangulateMultiview(tracks, camPoses, intrinsics);
    rng default;
    [~, camPoses] = bundleAdjustment(xyzPoints, tracks, camPoses, ...
        intrinsics, 'PointsUndistorted', true, 'AbsoluteTolerance', 1e-12, ...
            'RelativeTolerance', 1e-12, 'MaxIterations', 200, 'FixedViewID', 1);
    vSet = updateView(vSet, camPoses);
```

Le funzioni "findTracks" e "poses" estraggono dall'*imageviewset* rispettivamente la serie di coordinate dei singoli SURFPoints visti da più immagini e l'insieme delle pose calcolate fino ad

adesso. Questi dati vengono usati come input nella funzione "triangulateMultiview" che, come si può comprendere dal nome, triangola le coordinate spaziali di tutti i punti individuati in almeno due immagini sfruttando i dati di tutti i fotogrammi compresi dal primo a quello attuale.

Successivamente si applica "bundleAdjustment" per rifinire le pose in modo da minimizzare i *reprojection errors*. Tale funzione non è deterministica per cui si agisce sul Random Number Generator per uniformare i risultati.

```
%Identificazione delle posizioni e delle features dei punti proiettati.
%Accoppiamento delle features tra l'immagine corrente e quella
%precedente
[CPoints(:,:,viewId), currCFeatures, currCMatches, blacklist, ...
    blacklist_i] = helperDetectAndMatchFeaturesC(...
        prevCFeatures, Irgb, blacklist, blacklist_i, viewId);

%Nel caso si trovino solo 3 accoppiamenti tra i punti delle due immagini si
%collegano tra loro i punti non accoppiati
CMatches(1:size(currCMatches,1),:,viewId - 1) = currCMatches;
if size(currCMatches,1) == 3
    CMatches(4,:,viewId - 1) = [setdiff(1:4,CMatches(:,1,viewId-1)) ...
        setdiff(1:4,CMatches(:,2,viewId-1))];
end

if ~ismember(viewId ,blacklist)
    %Il primo fotogramma utile (j) viene determinato solo alla prima
    %immagine che non fa parte della blacklist
    if ~jDetermined
        for j = 1:viewId - 1
            if size(setdiff(1:j, blacklist),2) == 1
                jDetermined = true;
                break
            end
        end
    end

    %Riordinamento della tabella degli accoppiamenti affinché ogni riga
    %corrisponda sempre allo stesso punto per ogni immagine nonostante
    %matchFeatures possa assegnare valori differenti allo stesso punto
    whitelist = setdiff(j:viewId - 1, blacklist - 1);
    if size(whitelist, 2) > 1
        CMatches(:,:,whitelist(end)) = ...
            CMatches(CMatches(:,2,whitelist(end-1)),:,whitelist(end));
    end
end
```

Si utilizza "helperDetectAndMatchFeaturesC" come si è già fatto nel secondo fotogramma dopodiché, nel caso il fotogramma attuale non faccia parte della *blacklist*, si riordinano le righe della matrice *CMatches* appena ottenuta in modo che ogni riga corrisponda sempre allo stesso punto proiettato per qualsiasi fotogramma.

Per realizzare ciò si ricava qual è l'elenco delle matrici 4x2 all'interno di *CMatches* che mettono in relazione fotogrammi non appartenenti alla *blacklist*, tale matrice riga è stata chiamata *whitelist*.

Per ricavarla è prima necessario determinare qual è il primo fotogramma utilizzabile dal programma (ossia non appartenente alla *blacklist*) e si indica il suo *viewID* con *j:* dato che esso non varia nel corso del programma, prima di calcolarlo si verifica che non sia già stato calcolato in precedenza.

```
    %Preparazione per il fotogramma successivo
    prevI = I;
    prevFeatures = currFeatures;
    prevPoints   = currPoints;
    if ~ismember(viewId ,blacklist)
        prevCFeatures = currCFeatures;
    end
end
```

Prima di passare al fotogramma successivo si realizzano tutti i passaggi di variabili già applicati nella transizione tra il secondo e terzo fotogramma.

```
for viewId = windowSize + 1:numel(images.Files)
    %Lettura immagine
    Irgb = readimage(images, viewId);

    %Compensazione della distorsione
    I = undistortImage(rgb2gray(Irgb), intrinsics);
    Irgb = undistortImage(Irgb, intrinsics);

    %Creazione dei SURFPoint e delle features utili all'odometria nell'immagine
    %corrente. Accoppiamento delle features tra la prima e la seconda immagine
    [currPoints, currFeatures, indexPairs] = helperDetectAndMatchFeatures( ...
        prevFeatures, I);

    %Filtraggio dei punti usati per l'odometria
    inlierIdx = helperFindEpipolarInliers(prevPoints(indexPairs(:,1)), ...
        currPoints(indexPairs(:, 2)), intrinsics);
    indexPairs = indexPairs(inlierIdx, :);


    %Triangolazione dei punti trovati nell'immagine corrente usando i due
    %fotogrammi precedenti per determinare le loro coordinate mondo e
    %coordinate immagine (nel fotogramma attuale)
    [worldPoints, imagePoints] = helperFind3Dto2DCorrespondences(vSet, ...
        intrinsics, indexPairs, currPoints);

    %Si fa in modo che RANSAC possa realizare un numero di cicli maggiore
    %rispetto al massimo fissato
    warningstate = warning('off','vision:ransac:maxTrialsReached');

    %Estrazione di orientamento e posizione della posizione della
    %telecamera nell'istante attuale
    rng default;
    [orient, loc] = estimateWorldCameraPose(imagePoints, worldPoints, ...
        intrinsics);
```

```
%Ripristino dello stato di default
warning(warningstate)

%Aggiunta dei punti e degli accoppiamenti all'"imageviewset"
vSet = addView(vSet, viewId, rigid3d(orient, loc), 'Points', currPoints);
vSet = addConnection(vSet, viewId-1, viewId, 'Matches', indexPairs);
```

I processi per i fotogrammi successivi a *windowSize* procedono in modo identico al ciclo precedente fino a quando si deve applicare la rifinitura delle pose con la funzione "bundleAdjustment".

```
%Rifinitura delle posizioni della telecamera ogni "windowSize2" immagini
%per ridurre il carico di lavoro
if mod(viewId, windowSize2) == 0
    startFrame = max(1, viewId - windowSize);
    tracks = findTracks(vSet, startFrame:viewId);
    camPoses = poses(vSet, startFrame:viewId);
    [xyzPoints, reprojErrors] = ...
        triangulateMultiview(tracks, camPoses, intrinsics);

    %Mantenere fisse le prime due posizioni mantiene fissa la scala
    edIds = [startFrame, startFrame+1];

    %Filtraggio dei punti utilizzati nel bundle adjustment
    idx = reprojErrors < 2;

    %Rifinitura delle posizioni usando le ultime "windowSize" immagini
    rng default;
    [~, camPoses] = bundleAdjustment(xyzPoints(idx, :), tracks(idx), ...
        camPoses, intrinsics, 'FixedViewID', edIds, ...
            'PointsUndistorted', true, 'AbsoluteTolerance', 1e-12,...
                'RelativeTolerance', 1e-12, 'MaxIterations', 200);
    vSet = updateView(vSet, camPoses);
end
```

In questo ciclo la rifinitura delle pose viene applicata ogni *windowSize2* fotogrammi.

```
%Identificazione delle posizioni e delle features dei punti proiettati.
%Accoppiamento delle features tra l'immagine corrente e quella
%precedente
[CPoints(:,:,viewId), currCFeatures, currCMatches, blacklist, ...
    blacklist_i] = helperDetectAndMatchFeaturesC(prevCFeatures, Irgb, ...
        blacklist, blacklist_i, viewId);

%Nel caso si trovino solo 3 accoppiamenti tra i punti delle due immagini si
%collegano tra loro i punti non accoppiati
CMatches(1:size(currCMatches,1),:,viewId - 1) = currCMatches;
if size(currCMatches,1) == 3
    CMatches(4,:,viewId - 1) = [setdiff(1:4,CMatches(:,1,viewId-1)) ...
        setdiff(1:4,CMatches(:,2,viewId-1))];
end
```

```
    if ~ismember(viewId ,blacklist)
        %Il primo fotogramma utile (j) viene determinato solo alla prima
        %immagine che non fa parte della blacklist
        if ~jDetermined
            for j = 1:viewId - 1
                if size(setdiff(1:j, blacklist),2) == 1
                    jDetermined = true;
                    break
                end
            end
        end

        %Riordinamento della tabella degli accoppiamenti affinchè ogni riga
        %corrisponda sempre allo stesso punto per ogni immagine nonostante
        %matchFeatures possa assegnare valori differenti allo stesso punto
        whitelist = setdiff(j:viewId - 1, blacklist - 1);
        if size(whitelist, 2) > 1
            CMatches(:,:,whitelist(end)) = ...
                CMatches(CMatches(:,2,whitelist(end-1)),:,whitelist(end));
        end
    end

    %Preparazione per il fotogramma successivo
    prevI = I;
    prevFeatures = currFeatures;
    prevPoints   = currPoints;
    if ~ismember(viewId ,blacklist)
        prevCFeatures = currCFeatures;
    end
end
```

Anche la parte finale del ciclo procede in modo identico a quello precedente.

```
%Calcolo del fattore di scala
camPoses = poses(vSet);
scaleFactor = distanceCovered ./ ...
      sqrt(sum(camPoses.AbsolutePose(end).Translation.^2, 2));

R = camPoses.AbsolutePose(1).Rotation';
for i = 1:height(camPoses)
    %Applicazione del fattore di scala a tutte gli istanti della telecamera
    camPoses.AbsolutePose(i).Translation = ...
        camPoses.AbsolutePose(i).Translation * scaleFactor;

    %Aggiustamento delle matrici di rotazione di tutti gli istanti della
    %telecamera per far corrispondere l'orientamento iniziale ad una matrice
    %identità
    camPoses.AbsolutePose(i).Rotation = camPoses.AbsolutePose(i).Rotation * R;
end

%Aggiornamento dell'"imageviewset" dopo l'applicazione del fattore di scala
vSet = updateView(vSet, camPoses);
```

Una volta terminato il calcolo della traiettoria si ha la posa all'istante finale e si può quindi calcolare il percorso nelle dimensioni reali confrontando la distanza tra il punto iniziale e finale nella scala di default con la distanza effettiva fornita dall'operatore.

La funzione "bundleAdjustment" potrebbe aver ruotato la posizione iniziale della fotocamera, quindi la si riporta parallela all'asse z, aggiustando anche tutte le altre pose di conseguenza.

```
%Liste delle immagini e degli accoppiamenti tra fotogrammi che non fanno
%parte della blacklist
whitelist = setdiff(j:numel(images.Files), blacklist);
whitelistM = setdiff(j:numel(images.Files)-1, blacklist - 1);

%Creazione di "pointTrack" dei punti proiettati per l'applicazione della
%funzione "triangulateMultiview"
P1 = zeros(size(whitelist,2),2);
P2 = zeros(size(whitelist,2),2);
P3 = zeros(size(whitelist,2),2);
P4 = zeros(size(whitelist,2),2);
for i=1:size(whitelistM,2)
    P1(i,:) = CPoints(CMatches(1,1,whitelistM(i)),:,whitelist(i));
    P2(i,:) = CPoints(CMatches(2,1,whitelistM(i)),:,whitelist(i));
    P3(i,:) = CPoints(CMatches(3,1,whitelistM(i)),:,whitelist(i));
    P4(i,:) = CPoints(CMatches(4,1,whitelistM(i)),:,whitelist(i));
end
P1(end,:) = CPoints(CMatches(1,2,whitelistM(end)),:,whitelist(end));
P2(end,:) = CPoints(CMatches(2,2,whitelistM(end)),:,whitelist(end));
P3(end,:) = CPoints(CMatches(3,2,whitelistM(end)),:,whitelist(end));
P4(end,:) = CPoints(CMatches(4,2,whitelistM(end)),:,whitelist(end));
tracksC = [pointTrack(whitelist,P1), pointTrack(whitelist,P2), ...
    pointTrack(whitelist,P3), pointTrack(whitelist,P4)];

%Rifinitura delle coordinate mondo dei punti proiettati utilizzando tutte
%i fotogrammi non presenti nella blacklist
xyzCPoints = triangulateMultiview(tracksC, camPoses, intrinsics);
rng default;
[WorldCPoints, ~] = bundleAdjustment(xyzCPoints, tracksC, camPoses, ...
    intrinsics, 'PointsUndistorted', true, 'AbsoluteTolerance', 1e-12, ...
        'RelativeTolerance', 1e-12, 'MaxIterations', 200);
```

Si creano manualmente degli oggetti pointTrack (che in precedenza erano stati creati con la funzione "findTracks") per poter applicare "bundleAdjustment" anche ai punti proiettati.

Al contrario di ciò che si faceva per determinare la traiettoria, in questo caso si ignorano le pose rifinite che la funzione ci fornisce e si chiedono solo le coordinate dei punti colorati nello spazio.

```
%Inizio della riproduzione del video
Irgb = readimage(images, 1);
player = vision.VideoPlayer('Position', ...
    [0, 0, imageSize(1,2)+ 50, imageSize(1,1)+ 50]);
step(player, Irgb);
```

```
%Creazione del grafico in funzione della distanza dei punti proiettati
dim = max(abs(WorldCPoints), [],'all').*3./2;
figure
axis([-dim, dim, -dim, dim, -dim ./ 4, dim./4.*7]);

%Orientamento dell'asee y come verticale
view(gca, 3);
set(gca, 'CameraUpVector', [0, -1, 0]);
camorbit(gca, -120, 0, 'data', [0, 1, 0]);

%Intestazone del grafico
grid on
xlabel('X (cm)');
ylabel('Y (cm)');
zlabel('Z (cm)');
hold on
```

Terminata la fase di calcolo si può inizializzare la finestra che riprodurrà il video e si può creare il grafico tridimensionale su cui si disegnerà la traiettoria.

Anche i punti proiettati saranno raffigurati e, per far sì che essi siano visibili, si sceglie di rappresentare una porzione di spazio le cui dimensioni sono in funzione della posizione di tali punti.

```
%Plot della posizione iniziale, inizializzazione della traiettoria e
%rappresentazione dei punti proiettati
camEstimated = plotCamera(camPose, 'Size', dim ./ 25, ...
    'Color', 'g', 'Opacity', 0);
trajectoryEstimated = plot3(0, 0, 0, 'g-');
scatter3(WorldCPoints(:,1), WorldCPoints(:,2), WorldCPoints(:,3), [], 'r');

%Legenda del grafico
legend('Estimated Trajectory', 'Points');
title('Camera Trajectory');

%Le coordinate dei punti proiettati vengono esplicitate nella Command
%Window
disp(WorldCPoints)
```

In questa sezione si pone sul grafico la telecamera in posizione (0,0,0) e si raffigurano i punti proiettati.

Vengono inoltre stampate nella Command Window le coordinate di tali punti in modo che siano facilmente accessibili.

```
%Il programma viene messo in pausa prima di aggiornare il grafico e far
%partire il video, per iniziare la visione premere un qualsiasi tasto
pause;
```

La prossima sezione servirà ad aggiornare le immagini mostrate nel video player in contemporanea alla scrittura della traiettoria sul grafico; si mette quindi il programma in pausa in modo che l'utilizzatore sia pronto alla visione.

Per procedere si deve premere un qualsiasi tasto mentre il cursore è posizionato sulla Command Window di MatLab.

```
for viewId = 1:numel(images.Files)
    %Passaggio al fotogramma successivo
    Irgb = readimage(images, viewId);
    step(player, Irgb);

    %Aggiornamento della telecamera
    camEstimated.AbsolutePose = camPoses(viewId,2).AbsolutePose;

    %Aggiornamento della traiettoria
    locations = vertcat(camPoses{1:viewId,2}.Translation);
    set(trajectoryEstimated, 'XData', locations(:,1), 'YData', ...
        locations(:,2), 'ZData', locations(:,3));

    %Pausa di 0.25 secondi per rallentare la riproduzione del video
    pause(1/4);
end
```

Infine il programma aggiorna l'immagine mostrata nel video player e sposta la telecamera nel plot alla posa successiva ogni 0.25 secondi.

Per ulteriori informazioni sulle singole funzioni si rimanda alla documentazione del software MatLab[8].

3. Risultati

Il programma descritto fornisce in output la traiettoria e le coordinate dei punti di riferimento. Tali informazioni vengono poi graficate come riportato in **Figura 11**:

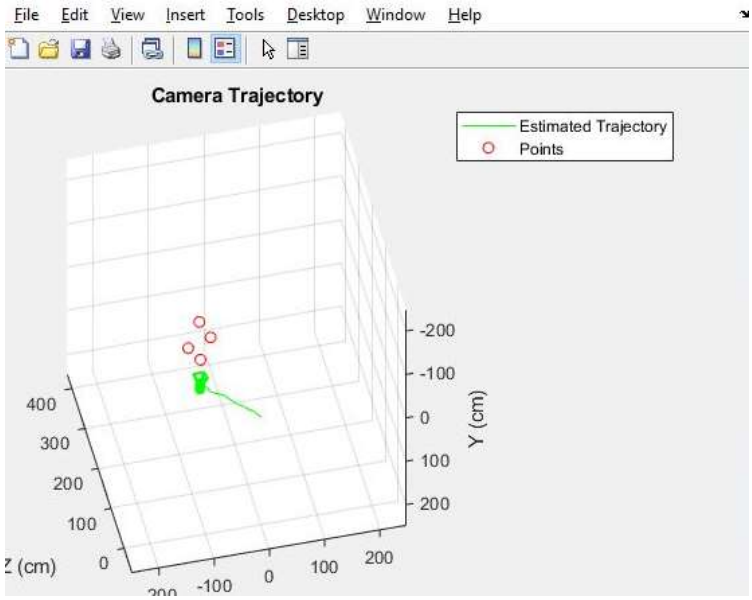

Figura 11: Esempio di output del programma

Come si è visto nel capitolo precedente, il percorso della telecamera viene calcolato non solo attraverso i quattro punti proiettati, ma anche riconoscendo tutti gli altri oggetti presenti nella ripresa.

Al fine di ridurre l'errore è quindi necessario riempire il video di quante più feature possibile per fare in modo che il programma abbia molti punti da triangolare. Nella maggior parte delle applicazioni però potrebbero non essere disponibili altri oggetti nei pressi della parete di cemento da analizzare, si può quindi pensare di proiettare un pattern di un secondo colore su una porzione non rilevante della superficie.

Si è testato quale fosse il numero minimo di oggetti in una ripresa per cui si potesse ottenere un risultato sufficientemente preciso ed accurato.

Una volta montata la telecamera sull'end-effector di un robot antropomorfo (**Figura 12**) si è fatto traslare lo strumento in direzione orizzontale, dalla posizione più a sinistra permessa dagli angoli di partenza fino al limite destro. Considerando il sistema di riferimento del grafico in **Figura 14**, ne è risultata una distanza percorsa di 74,8 cm nel verso positivo dell'asse x.

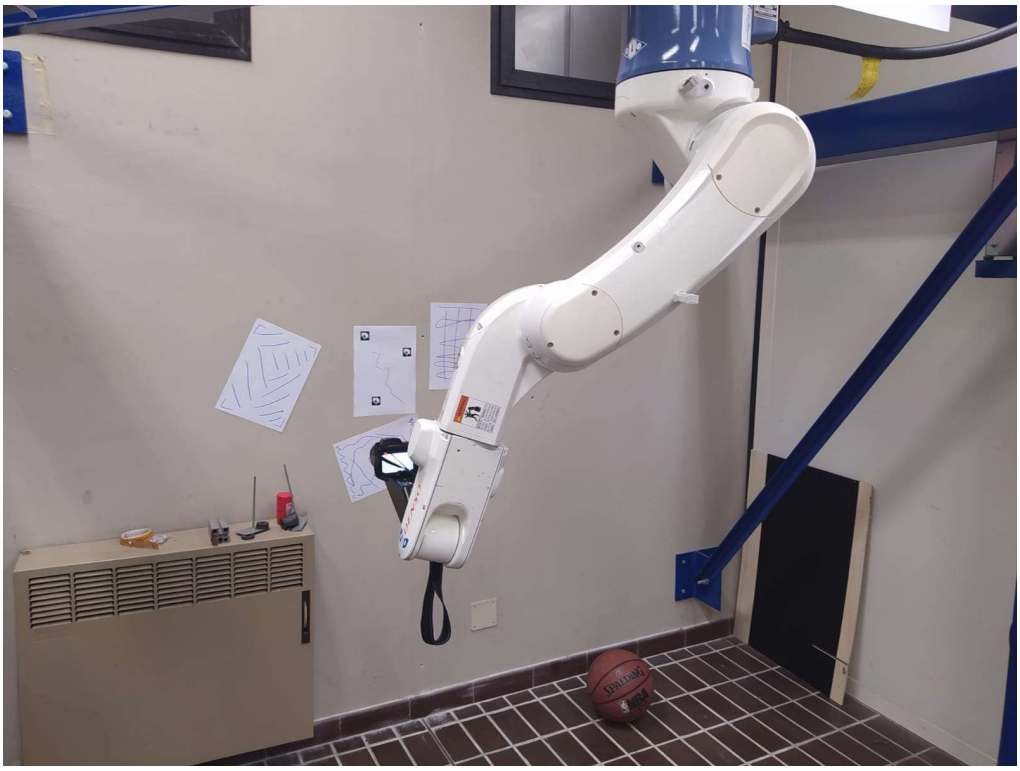

Figura 12: Braccio meccanico utilizzato

Si è ripetuto l'esperimento togliendo di volta in volta un numero maggiore di feature: nel primo tentativo, in cui sono presenti una palla da basket con un piedistallo e quattro fogli A4 incollati alla parete con delle feature disegnate (**Figura 13**), Il programma riesce a far coincidere esattamente la traiettoria calcolata (riportata in verde nella **Figura 14**) e la traiettoria effettivamente percorsa dal braccio meccanico (disegnata in rosso). All'interno della rappresentazione si riportano inoltre i SURFPoints (identificati col colore blu) che sono stati triangolati per ottenere la traiettoria.

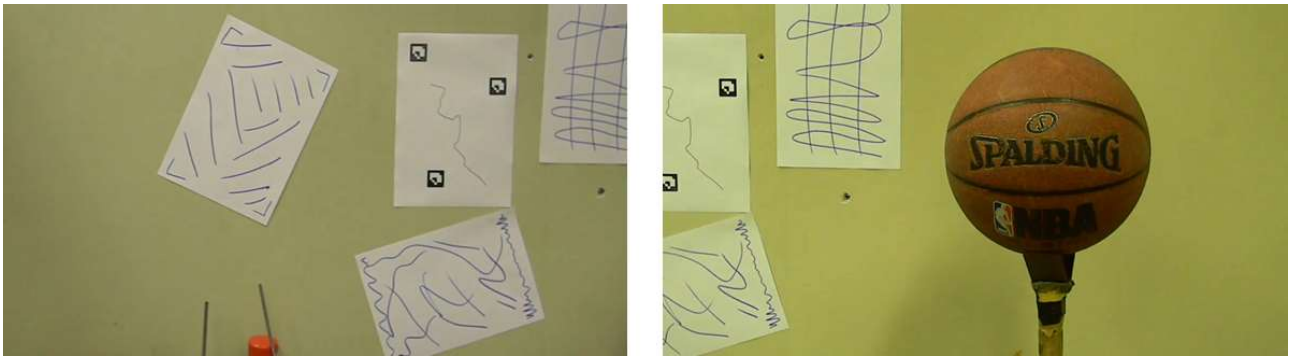

Figura 13: Immagini agli estremi della ripresa 1

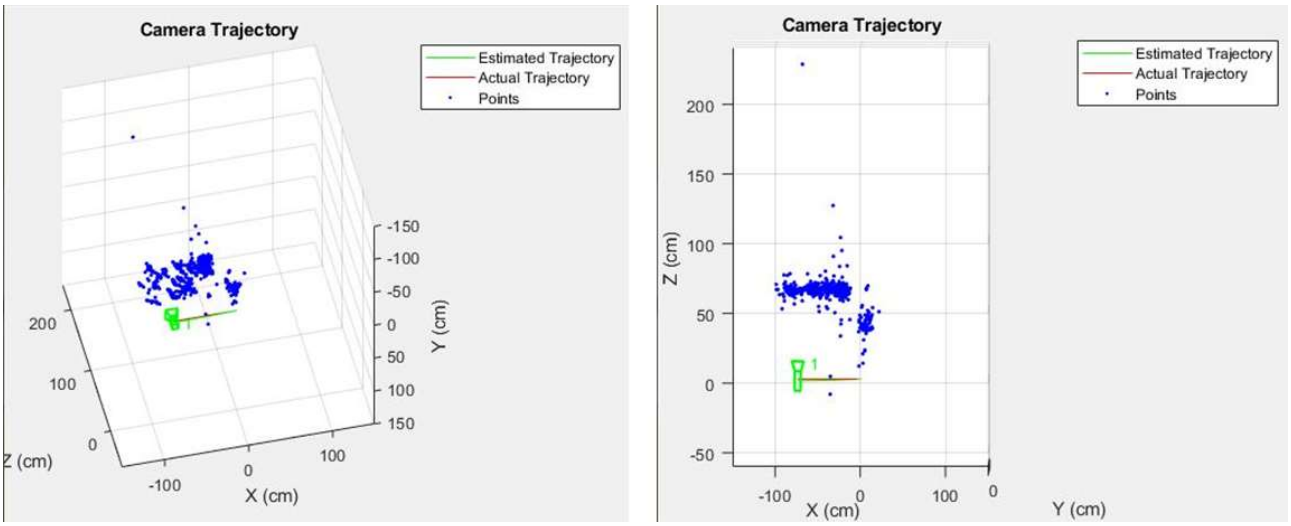

Figura 14: Due viste della traiettoria calcolata a partire dalla ripresa 1

Per la seconda prova si è rimosso il pallone dalla ripresa (**Figura 15**): si può notare che il programma non ha più a disposizione gran parte dei suoi riferimenti nel margine destro (**Figura 16**) ma, nonostante ciò, il riconoscimento della traiettoria avviene correttamente.

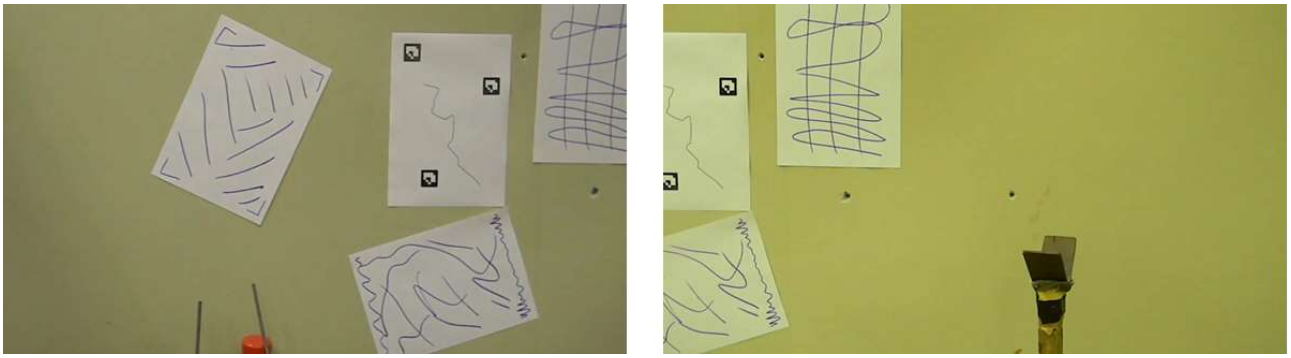

Figura 15: Immagini agli estremi della ripresa 2

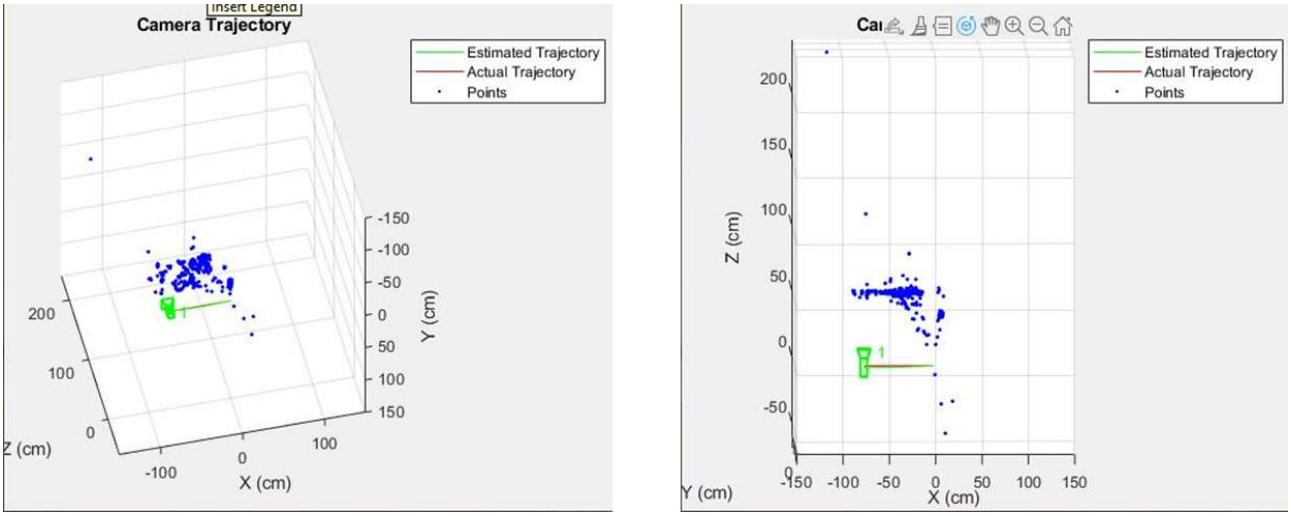

Figura 16: Due viste della traiettoria calcolata a partire dalla ripresa 2

Si è ripetuta la ripresa per una terza volta, togliendo in questo caso il piedistallo su cui si reggeva la palla: troviamo rappresentato qui sotto (**Figura 17**) il caso in cui il programma non è stato in grado di estrarre correttamente la traiettoria percorsa per mancanza di oggetti su cui basarsi.

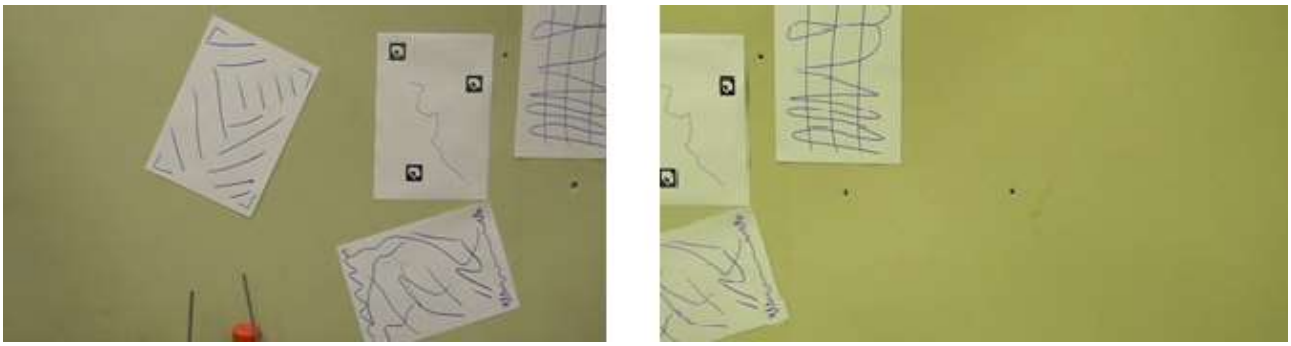

Figura 17: Immagini agli estremi della ripresa 3

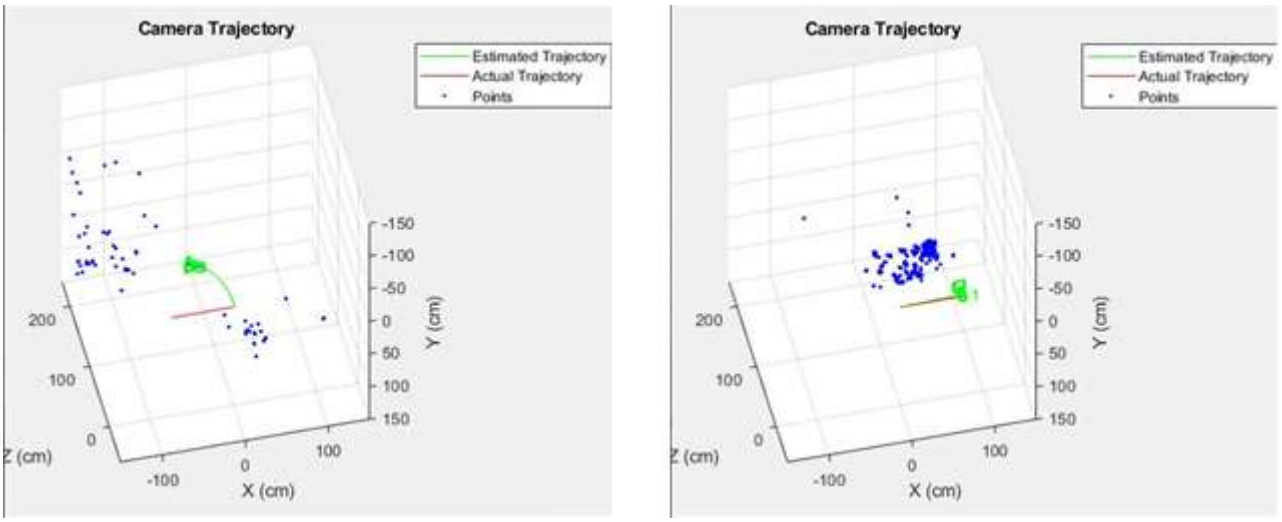

Figura 18: Traiettoria calcolata se ci si muove da destra verso sinistra oppure da sinistra verso destra

Nonostante questo esperimento sia stato di carattere qualitativo, esso ci permette di notare come è fondamentale riuscire a fornire al programma il maggior numero possibile di riferimenti nei primi fotogrammi.

Inserendo nel programma il video riprodotto al contrario si può notare come il risultato migliori notevolmente: ciò accade perché nel video riavvolto ci si sposta dall'immagine a sinistra verso l'immagine a destra e i fotogrammi con poche feature da riconoscere si trovano al termine del video; se invece si lasciano i fotogrammi con carenza di riferimenti nella parte iniziale della ripresa il programma non è inizialmente in grado di stimare correttamente le coordinate mondo dei punti osservati, portando a traiettorie erronee (**Figura 18**).

4. Conclusioni e sviluppi futuri

Il presente lavoro si inquadra in un ambito più ampio finalizzato all'identificazione di cricche superficiali in strutture in calcestruzzo (progetto EU H2020 ENDURCRETE – GA 760639). Per poter valutare in maniera oggettiva l'estensione delle cricche a partire da un'immagine, è però necessario effettuare una compensazione degli effetti prospettici attraverso la conoscenza della posa relativa tra il sistema di visione e la superficie target, nonché la distanza di almeno un punto tra detti sistemi. L'obbiettivo della presente tesi era pertanto quello di valutare l'efficacia di approcci di visual odometry monoculare per l'estrazione della posa relativa tra la camera e la superficie target. Si è sviluppato un software per l'elaborazione di immagini, partendo da codice già presente nell'ambiente Matlab Mathworks, che consente di estrarre tali informazioni e valutare la traiettoria percorsa dalla camera durante l'acquisizione di un filmato. Sono stati effettuati test per valutare l'effetto di immagini con diverse dotazioni di features nella scena, dimostrando che la presenza di numerosi dettagli è in realtà condizione fondamentale per incrementare la precisione e l'accuratezza nel riconoscimento dell'evoluzione della posa nel tempo.

Il confronto tra la posa estratta dal software e quella effettiva della camera è stato effettuato attraverso l'impiego di un robot antropomorfo che consentisse la realizzazione di traiettorie e pose definite. Come prevedibile, la precisione e l'accuratezza della ricostruzione della posa diminuiscono al diminuire di features nella scena inquadrata dalla camera.

Data la scarsa presenza di features nelle superfici target dell'applicazione per il riconoscimento delle cricche, è d'obbligo sottolineare che la tecnica indagata non si è rilevata adatta allo scopo. Si suggerisce, pertanto, di valutare l'impiego di tecnologie alternative come, ad esempio, le telecamere RGB+Depth.

.

5. Bibliografia

[1] Marco Piccin, Metodi di misura di distanza con laser.
https://digilander.libero.it/marpic/Frontespizio.pdf

[2] Paolo Medici, Elementi di analisi per Visione Artificiale
http://www.ce.unipr.it/people/medici/geometry/geometry.html

[3] Valentina Alena Girelli, Tecniche digitali per il rilievo, la modellazione tridimensionale e la rappresentazione nel campo dei beni culturali.
http://amsdottorato.unibo.it/310/

[4] Luca Grementieri, Geometria della visione.
https://amslaurea.unibo.it/8946/1/Grementieri_Luca_tesi.pdf

[5] Francesco Grossi, Stima del moto da sequenze di immagini omnidirezionali.
https://www.politesi.polimi.it/bitstream/10589/44362/1/2012_04_Grossi.pdf

[6] David Nistér. An efficient solution to the five-point relative pose problem. IEEE Trans. Pattern *Anal. Mach. Intell.*, 26(6):756-777, June 2004.
https://www-users.cs.umn.edu/~hspark/CSci5980/nister.pdf

[7] Martin Peris Martorell, Atsuto Maki, Sarah Martull, Yasuhiro Ohkawa, Kazuhiro Fukui, "Towards a Simulation Driven Stereo Vision System". Proceedings of ICPR, pp.1038-1042, 2012.
Sarah Martull, Martin Peris Martorell, Kazuhiro Fukui, "Realistic CG Stereo Image Dataset with Ground Truth Disparity Maps", Proceedings of ICPR workshop TrakMark2012, pp.40-42, 2012.
https://cvlab.cs.tsukuba.ac.jp

[8] Documentazione MatLab.
https://it.mathworks.com/help/