UNIVERSITÀ POLITECNICA DELLE MARCHE

FACOLTÀ DI INGEGNERIA

Corso di Laurea triennale in Ingegneria Meccanica

Applicazione della procedura SLAM (Simultaneous localization and mapping) per la determinazione di caratteristiche geometriche da file video

Mapping the environment from video file using SLAM (Simultaneous localization and mapping) process

Relatore: Chiar.mo
Paolo Castellini

Tesi di Laurea di:
Nicola Calcabrini

Correlatore: Chiar.ma
Milena Martarelli

Anno Accademico **2019** / **2020**

UNIVERSITÀ POLITECNICA DELLE MARCHE
CORSO DI LAUREA IN INGEGNERIA MECCANICA
FACOLTÀ DI INGEGNERIA
Via Brecce bianche – 60131 Ancona (AN), Italy

Alla mia famiglia e ai miei compagni di corso

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

"È facile navigare in spazi noti, ma in terreni sconosciuti?". Shweta Mayekar scrive così per dare un'idea sull'importanza della tecnologia SLAM, nell'articolo "the future of AR is SLAM technology, but what is SLAM?" del 2018.
L'auto a guida autonoma è forse il miglior esempio per spiegare l'ambito di utilizzo di questa tecnologia. La vettura deve essere in grado di creare la mappa 3D dei suoi dintorni nel più breve tempo possibile, per rispondere prontamente ai vari input che giungono. In generale ci troviamo di fronte al problema di ricreare un ambiente sconosciuto (strada, marciapiedi, persone, ostacoli) e, nel frattempo, tracciare la posizione del dispositivo (l'auto) che si muove all'interno di esso.

Questa tesi mira a comprendere il problema generale di SLAM e le sue principali tecniche di risoluzione con simulazioni effettuate attraverso MATLAB.

È suddivisa nei seguenti capitoli:

- *SLAM, teoria di base*: viene data una panoramica generale della tecnologia, con particolare attenzione al funzionamento mediante la piattaforma MATLAB.

- *Realizzazione nuvola di punti da file video*: si esaminano i passaggi richiesti per giungere all'estrazione di una nuvola di punti rappresentativa dell'ambiente, a partire da un file video.

- *Ottimizzazione nuvola di punti*: si comprendono i codici MATLAB utilizzati per l'eliminazione dei punti ritenuti inefficaci, con l'obbiettivo di ripulire la nuvola e ottimizzarla.

- *Creazione di superfici*: dalla nuvola di punti si generano delle superfici per riprodurre l'ambiente.

Capitolo 2

SLAM

2.1 Teoria di base

Un problema molto rilevante nella robotica è il cosiddetto problema di localizzazione simultanea (SLAM) dove l'obbiettivo è recuperare sia la traiettoria della telecamera sia la mappa dell'ambiente all'interno del quale essa si trova, sfruttando gli input provenienti dai dati del sensore.

Il problema della mappatura [1] consiste nell'utilizzo di un dispositivo consapevole della propria posizione assoluta che, per mezzo di dati e informazioni dei sensori, crea una mappa topologica dell'ambiente con cui interagisce.
Questo può essere trattato con diversi approcci divisibili in tre categorie:

- Future Based (FB): l'ambiente viene modellato per mezzo di informazioni relative a bordi e superfici. Il problema riguarda la stima della posizione e dell'orientamento, ma può anche includere colore e altre proprietà.
  Si tratta di un approccio rapido, utilizzato in particolare quando la mappatura deve essere eseguita in contesti esterni, con ampi spazi e pochi punti di riferimento.

- Occupancy Grid Based (OGB): lo spazio viene scomposto in una griglia di celle usate per determinare la presenza o meno di un ostacolo. Le informazioni sono maggiori dell'FB, ed è preferibile a quest'ultimo negli ambienti interni perché, in presenza di un certo numero di ostacoli, si ottiene la capacità di rappresentare oggetti di qualsiasi forma.

- Misto: è in grado di migliorare le prestazioni quando è necessaria la mappatura sia di ambienti con pochi punti di riferimento, sia di contesti interni. Ad ogni passo, in base alle misure registrate, lo stato viene aggiornato e può essere implementato uno dei precedenti approcci.

Nel problema di localizzazione, è invece resa disponibile una mappa, al fine di ottenere la posizione del dispositivo interagendo con essa. I principali metodi di risoluzione sono:

- Triangulation: sfrutta la relazione geometrica tra il dispositivo mobile e i punti di riferimento.

- Fingerprinting: si basa sulla corrispondenza delle "impronte digitali" misurate dal dispositivo mobile, con quelle del database, per individuare e tracciare il dispositivo.

- Proximity: viene implementata una fitta griglia di dispositivi di riferimento, ognuno con un proprio raggio di azione, che crea connessioni con il dispositivo mobile permettendo la localizzazione di quest'ultimo.

- Self-measurements: le auto-misurazioni vengono raccolte da un dispositivo mobile dotato di un'unità di misura inerziale (IMU). Questo è composto da un giroscopio per la misura della velocità angolare, e un accelerometro 3D che fornisce l'accelerazione lineare. Data la posizione iniziale è dunque possibile rilevare la localizzazione in ogni istante.

SLAM (Simultaneous Localization And Mapping) si riferisce al processo di calcolo della posizione e orientazione della telecamera rispetto ai suoi dintorni, mappando allo stesso tempo l'ambiente, solo per mezzo di input visivi. Apparentemente può sembrare che le due azioni, precedentemente descritte, dipendano l'una dall'altra, con l'impossibilità dunque di avviare il processo, ma ci sono vari algoritmi noti per risolverlo.

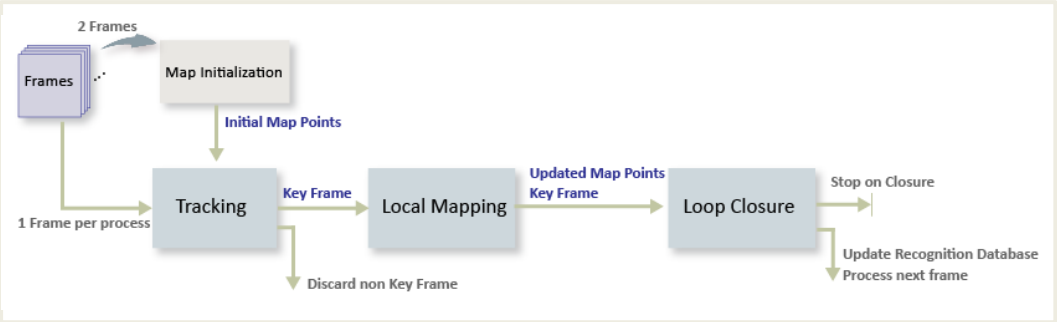
Le soluzioni a questo problema sono utilizzate in varie applicazioni, tra cui: autovetture a guida autonoma, rover planetari, robot domestici, veicoli subacquei autonomi, realtà aumentata. L'impiego di soluzioni basate sulla tecnologia SLAM permette di ottenere efficienti nuvole di punti in tempo reale.

In sintesi, la tecnologia SLAM esegue il processo per cui un dispositivo che si muove in un ambiente sconosciuto costruisce "in tempo reale" la mappa di tale ambiente.

2.2 ORB-SLAM

Questa tesi pone particolare attenzione al processo ORB-SLAM: un algoritmo SLAM completo per fotocamere monoculari. Il sistema funziona in tempo reale con CPU standard in una vasta gamma di ambienti, dalle piccole sequenze interne, ai droni che volano in spazi aperti, alle auto che guidano in città.

Il processo prende in input una sequenza di immagini estratte da un video. Nella prima fase avviene la Map Initialization, cioè l'inizializzazione della mappa dei punti 3D, a partire da due frames, poiché la profondità non può essere creata da una singola immagine. I punti 3D e la relativa posizione della telecamera vengono calcolati usando un metodo di triangolazione. Si passa quindi al Tracking: per ogni fotogramma si cercano corrispondenze con la key frame precedente, stimando la posizione della telecamera e monitorando la mappa.
Durante il Local Mapping vengono selezionate le immagini chiave, con le quali si aggiungono nuovi punti alla mappa, e si passa infine alla fase del Loop Closure per rivelare e chiudere il ciclo.



[Fig.1] Schema del processo SLAM.
Sono rappresentate le 4 fasi principali
del processo, e il loro ordine di utilizzo.

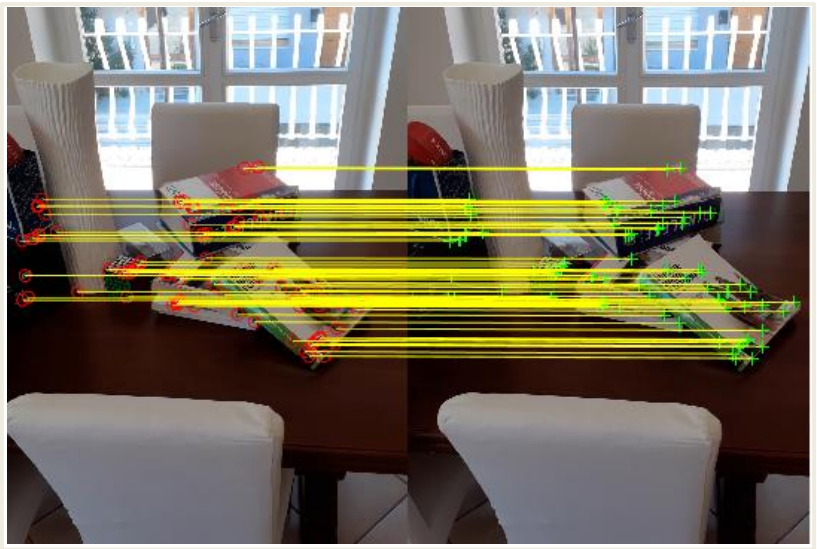
Il processo è dunque divisibile in quattro step principali [2]:

- Map Initialization: il programma sceglie due fotogrammi e ne determina la posizione relativa. Viene anche effettuata la triangolazione di una serie iniziale di punti della mappa.
  Per la ricerca di corrispondenze [3] ci viene in aiuto la funzione *matchFeatures,* per poi passare al calcolo in parallelo di due modelli geometrici:
  - Homography: processata tramite la funzione *estimateGeometricTransform*, viene utilizzata nel caso di scene planari.
  - Fundamental Matrix: per la quale si utilizza *estimateFundamentalMatrix,* per scene non planari.

  Il modello che provoca un errore di proiezione minore viene selezionato per stimare la rotazione e la traslazione relative tra i due fotogrammi utilizzando *relativeCameraPose.*

  Questi due fotogrammi sono le prime key frames trovate dal programma, alle quali saranno aggiunte tutte le successive. Per key frame si intende un fotogramma che contiene informazioni per la localizzazione e per il tracciamento. Due

fotogrammi chiave consecutivi di solito comportano sufficienti cambiamenti visivi.
In particolare, nel caso che andremo ad analizzare, l'inizializzazione della mappa si ha con le immagini 1 e 16, scelte da MATLAB come coppia tale da avere una stima della localizzazione dei punti 3D in comune, e della posizione relativa della telecamera. E sono le prime su un totale di 107 fotogrammi chiave.



[Fig.2] Map Initialization.
Nel caso analizzato l'inizializzazione avviene con
le frames 1 (a destra) e 16 (a sinistra).
Ben visibili i punti in comune rilevati dal programma.

- Tracking: si tratta del monitoraggio eseguito con ogni frame per l'ottimizzazione della posizione della telecamera. Se il tracciamento ha avuto esito positivo per l'ultimo fotogramma, si prevede la posizione del dispositivo tramite un modello di movimento a velocità costante. Se il tracciamento viene perso si fa affidamento al database delle key frames per la rilocalizzazione globale.

In questo step è necessaria l'analisi di ogni fotogramma, alcuni dei quali saranno selezionati come key frame. Il tutto termina una volta trovata una chiusura ad anello.

Nel processo di tracciamento si utilizzano diverse funzioni, riportate in ordine di utilizzo:

- *matchFeatures*: per ogni nuovo fotogramma cerca corrispondenze con l'ultima key frame.

- *estimateWorldCameraPose*: stima la posizione della videocamera:

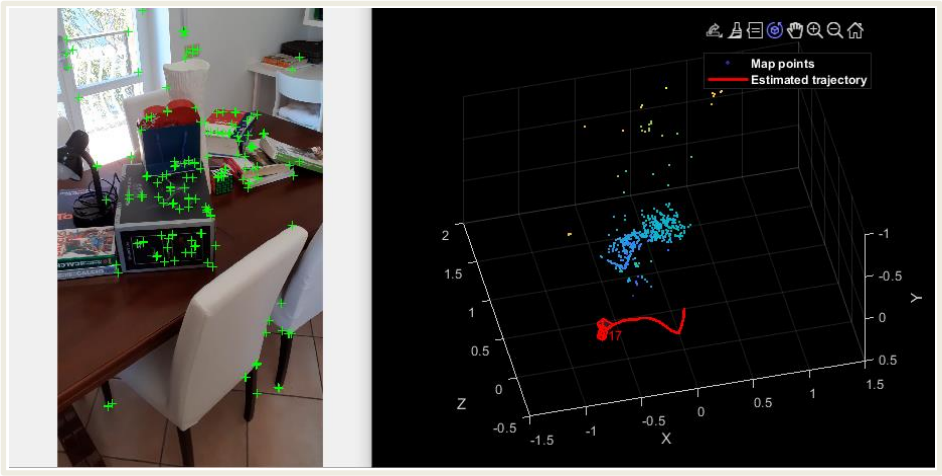
- *helperMatchFeaturesInRadius*: cerca corrispondenze tra i punti della mappa osservati nell'ultima key frame e l'immagine corrente.

- *bundleAdjustmenteMotion*: perfeziona la traiettoria basandosi sulla corrispondenza da 3D a 2D nel fotogramma corrente.

- *helperMatchFeaturesInRadius*: proietta i punti della mappa locale nel frame corrente per cercare più corrispondenze.

- *bundleAdjustmentMotion*: perfeziona nuovamente la posizione del dispositivo.

Infine, il programma decide se il fotogramma corrente è una nuova key frame. In caso lo fosse continua il processo di mappatura locale, altrimenti si passa al tracking del fotogramma successivo.



[Fig.3] Tracking.
A sinistra l'immagine corrente che il programma sta analizzando.
A destra la mappa dei punti trovati e la stima della traiettoria della telecamera con il numero della key frame corrente (17).

- Local Mapping: viene eseguita per ogni key frame. Prevede l'aggiornamento dei punti della mappa, che avviene quando si determina un nuovo fotogramma chiave.
  Per garantire il minor numero possibile di valori anomali, è necessario osservare un punto della mappa valido in almeno tre di questi.

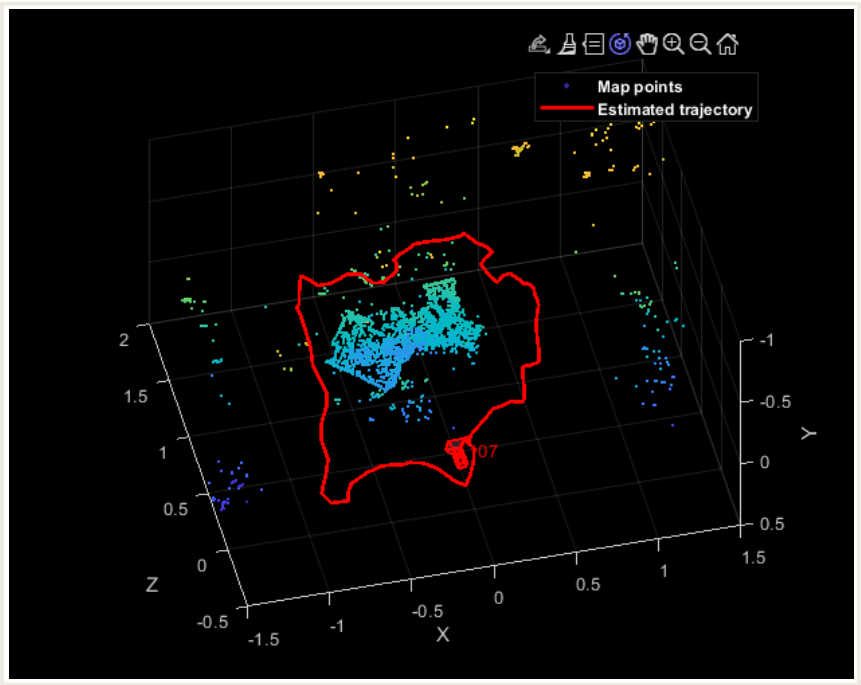
  La creazione di nuovi punti avviene tramite la triangolazione dei punti caratteristici della key frame corrente e in quelli collegati. Nel caso di punti senza eguali nel fotogramma corrente viene cercata una corrispondenza in altre key frames collegate, utilizzando la funzione *matchFeatures*.

  Avviene, tra l'altro, un perfezionamento della posizione del fotogramma chiave corrente, di quelli collegati e dei punti della mappa osservati in questi.

- Loop Closure: dall'attuale fotogramma chiave elaborato nel processo di mappatura locale si tenta di rilevare ed eseguire la chiusura del ciclo.

  La funzione *assessImageRetrieval* individua nel database le immagini simili al fotogramma chiave corrente. Una key frame candidata è valida se non è collegata all'ultima key frame, e tre dei fotogrammi chiave vicini sono candidati loop.

  Individuato un candidato loop valido, si determina la posizione relativa tra questo e la key frame corrente, si aggiunge la connessione ad anello e si aggiornano *mapPointSet* e *vSetKeyFrames*.



[Fig.4] Fine processo.
Sono stati individuati tutti i punti e il processo
si chiude alla key frame numero 107.

Capitolo 3

Realizzazione nuvola di punti da file video

3.1 Estrazione Parametri

Il processo SLAM che andremo ad analizzare prende in input un video girato in un luogo chiuso, con risoluzione 1920x1080 fullHD. Crea quindi una nuvola di punti attraverso la quale viene modellata una superficie che riproduce l'ambiente.
Il codice legge il video, dividendolo in molteplici immagini. Queste vengono analizzate una ad una per trovare la coppia di immagini chiave iniziali. L'inizializzazione della mappa parte dalle prima due key frames trovate, cioè immagini che contengano abbastanza informazioni e sufficienti differenze da consentire la localizzazione del dispositivo.

Molte funzioni del programma richiedono però la conoscenza di parametri intrinseci della telecamera. Tra questi, in particolare, sono necessari:

- *focal length*: è una misura della forza [4] con cui un sistema ottico converge o diverge la luce. Una lunghezza focale positiva indica che un sistema converge la luce, mentre una lunghezza focale negativa indica che la diverge.

  Tale grandezza [5] viene specificata come vettore di due elementi: [f_x; f_y].

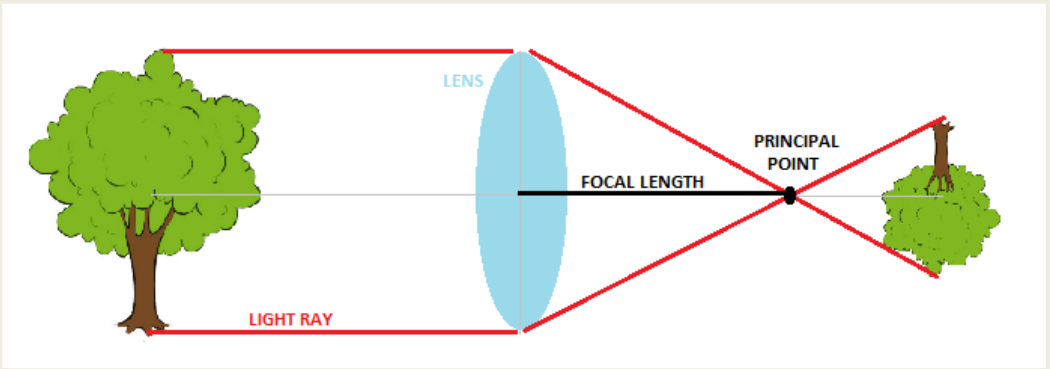
  $$F_x = F \times s_x$$
  $$F_y = F \times s_y$$

In cui:

- F è la lunghezza focale in millimetri.
- [s_x, s_y] sono il numero di pixel per millimetri rispettivamente in direzione x e y.
- f_x e f_y sono in pixel

- *principal point*: è il punto [6] in cui i piani principali, che determinano l'ingrandimento del sistema, attraversano l'asse ottico. È anche il punto da cui viene misurata la lunghezza focale dell'obbiettivo.

  Si tratta quindi del centro ottico della telecamera, specificato come vettore di due elementi, [c_x; c_y], in pixel.



[Fig.5] Parametri intrinseci.
Rappresentazione grafica dei parametri
focal length e *principal point*.

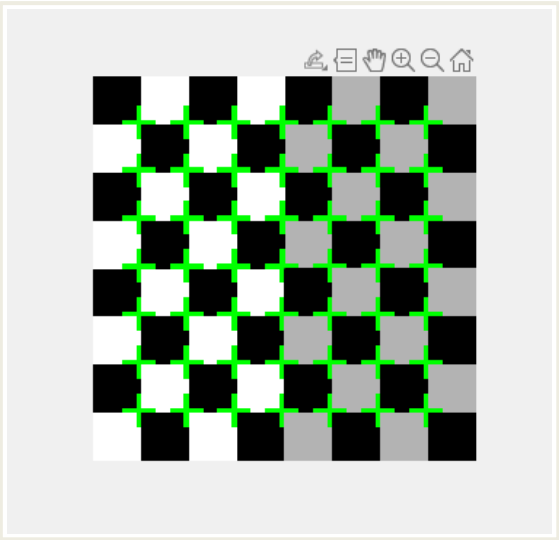
I parametri vengono estratti direttamente all'interno di MATLAB tramite un'app integrata: Camera Calibrator. Questa consente di stimare i parametri intrinseci, estrinseci e della distorsione dell'obbiettivo della fotocamera.
Per la stima è necessario disporre di punti del mondo 3D e dei corrispondenti punti di immagini 2D. È possibile ottenere queste corrispondenze utilizzando più immagini di un modello di calibrazione, nel nostro caso una scacchiera.

Il procedimento richiesto da Camera Calibrator è inizialmente di tipo manuale:

1. Il primo step prevede la creazione della scacchiera. Questo è possibile farlo direttamente dall'editor di MATLAB, runnando il seguente codice:

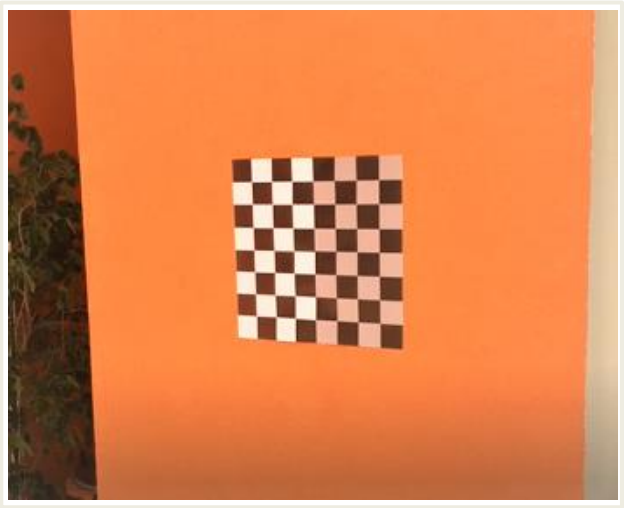
```
I = checkerboard;
squareSize = 10;
worldPoints = generateCheckerboardPoints([8 8], squareSize);
imshow(insertMarker(I, worldPoints + squareSize));
```

Si genererà una scacchiera 8x8, con metà quadrati bianchi e neri, e l'altra metà grigi e neri.



[Fig.6] Scacchiera generata.
Le croci verdi all'incrocio dei quadratini scompaiono in seguito alla stampa.

2. La scacchiera generata va quindi stampata, così da poterla usare come soggetto del video registrato dal dispositivo. Noi abbiamo scelto per la stampa un formato A2 (ottenuto unendo due fogli A3), preferibile ad un più comune A4 per la migliore risoluzione e, quindi, la maggiore affidabilità del risultato. Deve essere in seguito fissata su una parete lontano, per quanto possibile, dalla presenza di oggetti o colori che potrebbero interferire con l'analisi.



[Fig.7] Scacchiera stampata.
Abbiamo fissato la scacchiera su una parete spoglia per evitare interferenze, e con spazio attorno per permettere libertà di movimento durante la ripresa.

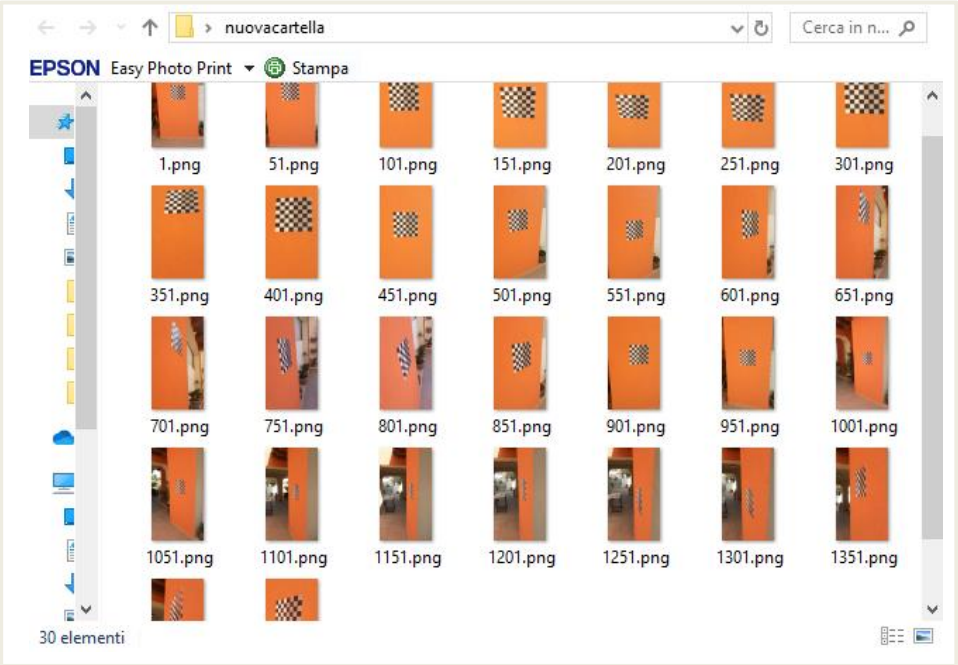
3. Come già anticipato, di questa scacchiera bisogna ora fare un video con il dispositivo di cui si vogliono estrarre i parametri. Questo deve essere tale che il soggetto venga ripreso da più angolazioni, così da assicurare all'app tutti i dati necessari.

[Fig.8] Ripresa scacchiera.
Durante il video la scacchiera viene ripresa
da più angolazioni, come quella in figura.

4. Camera Calibrator non lavora direttamente sul video, quindi questo deve essere prima letto e diviso in frames. Alcune di queste saranno salvate e utilizzate nell'app.
   Nel nostro caso il video, della durata di 49 secondi, viene frammentato in 1499 immagini, 30 delle quali abbiamo scelto di salvare in una nuova cartella.
   È possibile effettuare il tutto tramite il seguente codice:

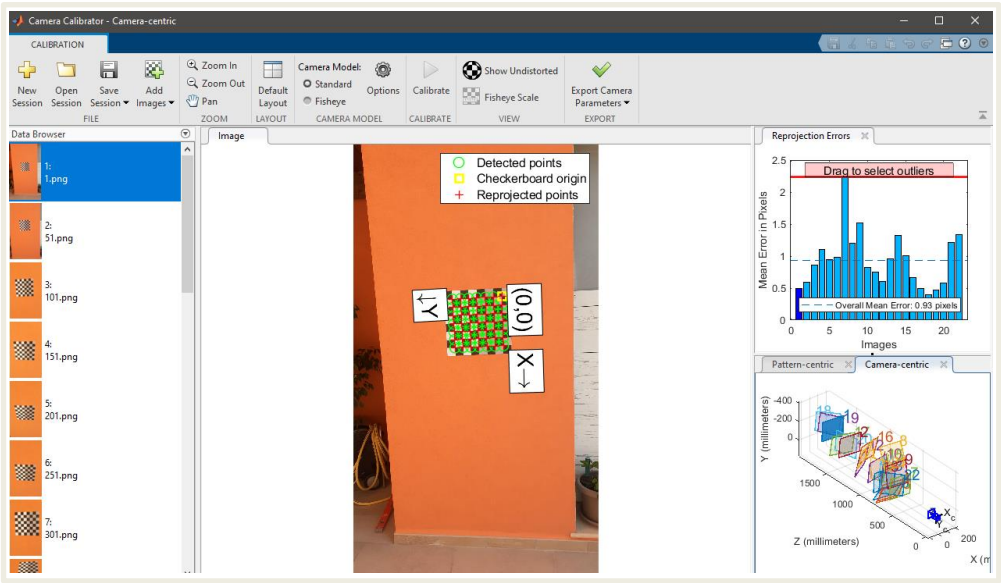
```
dataFolder      = 'C:\Users\User\Desktop\tirocinio\SLAM\';
dirname = 'C:\Users\User\Desktop\nuovacartella\';
currFrameIdx = 1;
while currFrameIdx < 1499
   vid = VideoReader(fullfile(dataFolder,'videoscacchiera.mp4'));
   numFrames = vid.NumberOfFrames;
   currI = read(vid,currFrameIdx);
   himage = imshow(currI);
   fname_app=[dirname '\' num2str(currFrameIdx) '.png']
   imwrite(currI, fname_app)
   imds          = imageDatastore(dirname);
   currFrameIdx = currFrameIdx+50
end
```

Il codice prevede un ciclo while con il quale leggere l'intero video *videoscrivania.mp4*, dalla prima all'ultima immagine. Seleziona poi una di queste ogni 50 lette, e le salva nella *nuovacartella*. Alla fine del processo saranno presenzi, come desiderato, 30 immagini della scacchiera ripresa da diverse angolazioni.



[Fig.9] Salvataggio foto scacchiera.
Sono state salvate le 30 immagini, una ogni 50 a partire dalla prima, come si può notare dai numeri.

5. Ci si affida ora a Camera Calibrator. L'app richiede in input le immagini salvate e la grandezza in centimetri dei quadratini della scacchiera stampata (4.8 nel caso analizzato), per ricavare i parametri del dispositivo.
   Alcune delle immagini saranno eliminate perché ritenute inutili dal programma. In particolare, delle 30 salvate ne saranno utilizzate 23.



[Fig.10] Camera Calibrator.
L'app riporta le frames utilizzate (a sinistra), un grafico dei valori anomali con i rispettivi errori in pixel per ogni immagine (in alto a destra) e l'orientazione di ogni ripresa rispetto alla telecamera (in basso a destra).

Terminato il processo di calibrazione, i dati devono essere aperti generando uno script di MATLAB (opzione presente nell'app). Al codice generato vanno aggiunte le seguenti righe:

```
[params,~,errors] = estimateCameraParameters(imagePoints,...
                    worldPoints, 'ImageSize',imageSize);

displayErrors(errors,params);
```

Il running dell'editor permette di leggere, tra l'altro, i parametri cercati:

Focal length (pixels): [1525.7405 +/- 5.7463 1493.5099 +/- 5.4326]
Principal point (pixels): [517.7645 +/- 2.7065 977.6792 +/- 3.6138]

Questi andranno inseriti, a meno dell'errore, nel codice principale per l'avvio del processo di inizializzazione.

3.2 Isolamento frames

Il punto di partenza di questo progetto è stato un codice MATLAB che eseguiva il processo SLAM a partire da frames sequenziali. Queste erano state selezionate e salvate da un precedente video ad una certa frequenza.
Dopo aver ispezionato la prima immagine, il programma esaminava le successive in cerca di corrispondenze, così da poter avviare il processo di inizializzazione della mappa. Dalla coppia di immagini trovata venivano stimate le loro relative rotazioni e traslazioni. Date la posizione della telecamera e i punti caratteristici abbinati, si determinava la localizzazione 3D dei punti usando una funzione di triangolazione.

Nel passaggio dalla lettura di immagini a quella del video sorge però un problema: non c'è più una cartella di appoggio delle immagini come invece accadeva nel caso di caricamento di frames sequenziali. Il codice analizza le immagini direttamente attraverso la lettura del video.
La presenza di un database in cui raccogliere le key frames è indispensabile nella fase di tracking: se viene meno il riconoscimento della posizione rispetto all'immagine chiave precedente, si fa affidamento al database per una rilocalizzazione globale calcolando le corrispondenze con ciascuna key frame precedente.

Per ovviare a questo problema è stata necessaria la creazione di una cartella temporanea, per permettere il salvataggio al suo interno delle immagini chiave trovate.

Di seguito sono riportate le prime righe del codice principale nel caso di lettura video:

```
dataFolder      = 'C:\Users\User\Desktop\tirocinio\SLAM\';
dirname = 'C:\Users\User\Desktop\tirocinio\SLAM\temporanea';
currFrameIdx = 1;
vid = VideoReader(fullfile(dataFolder,'videoscrivania4.mp4'));
numFrames = vid.NumberOfFrames;
currI = read(vid,currFrameIdx);
himage = imshow(currI);
fname_app=[dirname '\' num2str(currFrameIdx) '.png']
imwrite(currI, fname_app)
imds          = imageDatastore(dirname);
```

Il video, la cui collocazione nel computer è indicata da *dataFolder*, viene letto tramite la funzione *VideoReader* e diviso in frames. Queste vengono salvate in una cartella inizialmente vuota localizzata da *dirname*. La funzione *fname_app* permette il salvataggio di ogni key frame sotto uno specifico nome in formato ".png".

3.3 Creazione nuvola di punti

La fase di inizializzazione della mappa prevede, tra l'altro, la triangolazione di una serie iniziale di punti che determinano le corrispondenze tra le prime due key frames trovate. Questi vengono salvati su una mappa generata da MATLAB, in cui è riportata anche l'orientazione e la traccia della telecamera.
Durante la fase di tracking la mappa si aggiorna ad ogni immagine analizzata, con la possibilità di osservare il moto e la rotazione del dispositivo. I punti vengono aggiunti solo quando l'immagine corrente viene definita come chiave per il processo. Trovata una nuova key frame, contenente informazioni su localizzazione e tracciamento con sufficienti cambiamenti visivi, la nuvola si aggiorna con nuovi punti. Un nuovo punto è ritenuto valido se viene osservato in almeno tre key frames.

Capitolo 4

Ottimizzazione nuvola di punti

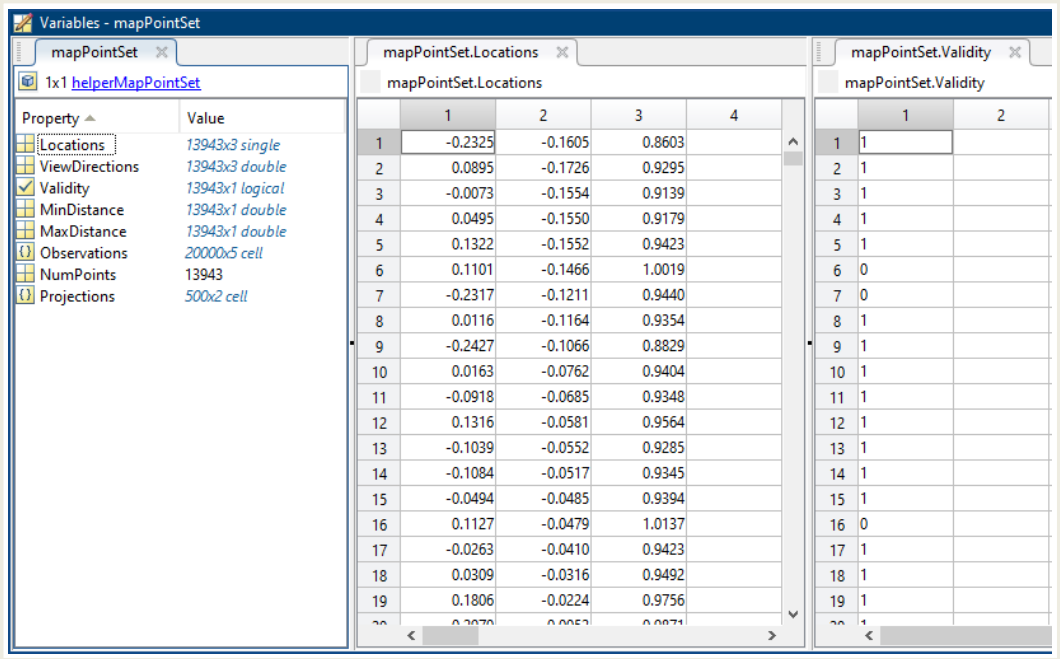
4.1 Filtraggio dei punti

Come anticipato nel precedente capitolo, i punti vengono aggiunti alla nuvola quando il sistema rileva delle corrispondenze tra le key frames.

Le informazioni riguardanti i punti trovati [7] sono contenute all'interno del *mapPointSet*, per la gestione dei dati delle mappe. Quest'oggetto viene utilizzato per memorizzare gli attributi dei punti della mappa. Al suo interno sono presenti in particolare:

- *Locations*: una matrice a 3 colonne all'interno della quale sono riportate le coordinate nello spazio di ogni punto.
- *ViewDirections*: anch'essa matrice a 3 colonne, rappresenta la direzione di vista di ogni punto.
- *Validity*: ad ogni punto viene assegnato un valore, 1 o 0, in base al fatto che il sistema lo riconosca come valido o meno.
- *MinDistance*: rappresenta la minima distanza invariante per ogni punto.
- *MaxDistance*: rappresenta la massima distanza invariante per ogni punto.
- *Observations*: una matrice a 5 colonne che indica le corrispondenze tra lo spazio 2D e 3D.
- *NumPoints*: riporta il numero totale dei punti trovati, coincidente quindi con il numero di righe delle matrici precedentemente descritte.

La rappresentazione di quanto descritto è riportata nella seguente figura:



[Fig.11] mapPointSet.
A sinistra è riportato l'intero oggetto *mapPointSet* e le matrici presenti al suo interno. Al centro la matrice *Locations* con le coordinate dei primi 19 punti. A sinistra la matrice *Validity* la validità dei primi 19 punti.

Il processo con cui il sistema raccoglie nuovi punti non è del tutto affidabile, tant'è che esso stesso riconosce la non validità per alcuni di questi. Non tutti i punti validi inoltre sono utili o efficaci: se ne possono trovare alcuni esterni, appartenenti ad oggetti posti fuori dall'ambiente che si desidera rappresentare. È necessario quindi effettuare un processo di scrematura dei punti. L'obbiettivo è eliminare tutti quelli con validità nulla e quelli esterni all'ambiente principale.

4.1.1 Validity

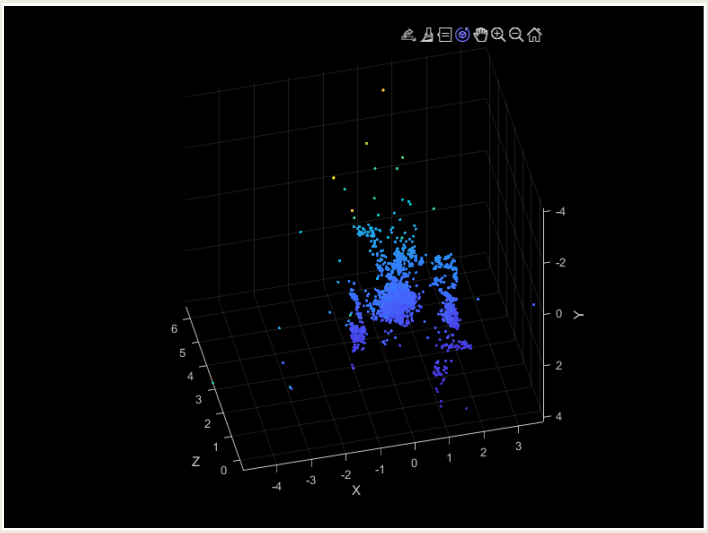
Per lasciare inalterate le matrici del *mapPointSet*, è stata creata una nuova matrice, denominata *pointok*, su cui disporre solo i punti con validità non nulla.
Il codice creato prevede l'esaminazione di ogni punto della Location in ordine, a partire dal primo. Nel momento in cui alle coordinate presenti nella riga *jj* corrisponde, nella stessa riga della *Validity*, una validità pari a uno, il rispettivo punto viene riportato in *pointok*.
Se la validità risulta essere nulla, il codice passa al punto successivo.

```
count=1;
   for jj=1:length(mapPointSet.Validity)
    if mapPointSet.Validity(jj)==1
    pointok(count,:)=mapPointSet.Locations(jj,:);
    count=count+1;
    else
    end
end
```

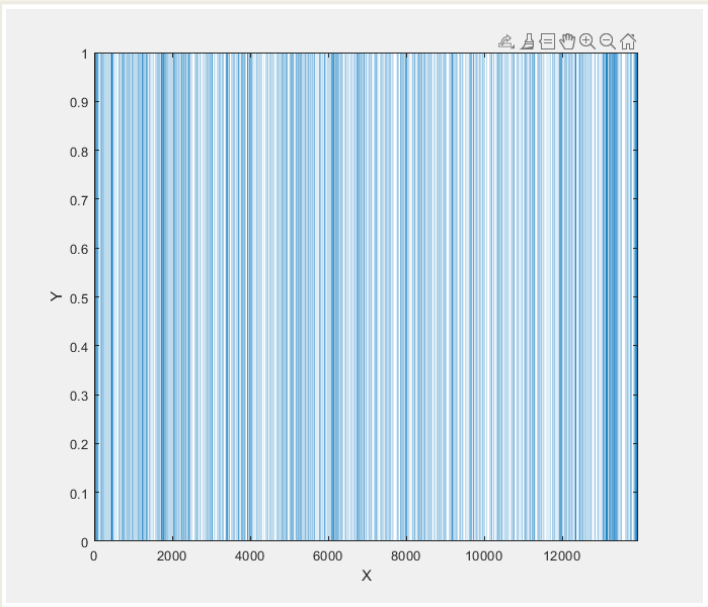
Nel nostro caso, come visibile in Fig.11, il numero totale dei punti trovati è 13.943, indicato dal *NumPoints*. La percentuale di punti validi, *NumValidPoint*, viene mostrato nel Command Window ed è ricavato tramite il seguente codice:

```
numValidPoint=sum(mapPointSet.Validity)…
                        /length(mapPointSet.Validity)*100
```

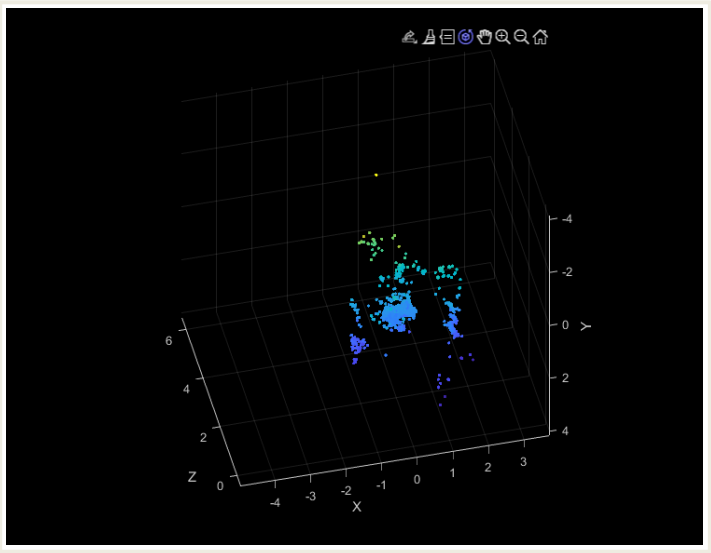
In cui la somma dei valori del *mapPointSet.Validity* è anche il numero di punti con validità pari a 1. Nel caso analizzato la percentuale è di 37,7752%, da cui ricaviamo che i punti validi sono 5267.



[Fig.12] Nuvola punti totali.
Il grafico riporta tutti i punti trovati dal programma (13.943) ovvero tutti quelli presenti nel *mapPointSet*.



[Fig.13] Percentuale punti validi.
Il grafico a barre riporta in X i punti totali (13.943), in Y il valore della validità. Il grafico risulta ovviamente a barre omogenee alternate perché la validità può essere solo 1 o 0.



[Fig.14] Nuvola punti validi.
Il grafico riporta solo i punti con validità pari a 1 (5267), ovvero tutti quelli attualmente presenti nella matrice *pointok*.

4.1.2 Punti esterni

Tra i punti validi, ce ne sono alcuni appartenenti a oggetti situati fuori dall'ambiente. Questi non fanno parte del soggetto che si vuole analizzare e, per tale motivo, sono denominati "punti esterni". Nella creazione della superficie questa comprenderebbe tutti i punti, anche quelli esterni, a causa dei quali risulterebbe distorta. Questi punti vanno dunque eliminati, così da isolare la nuvola dell'ambiente da riprodurre.

Per fare ciò abbiamo ideato un codice che prende in esame ogni punto, la cui posizione nella nuova matrice è data dal valore *k*, e ne misura la distanza con ogni altro punto della stessa matrice, in posizione *i*.

Quindi entrambe le variabili hanno inizialmente valore unitario, e valore massimo coincidente con la lunghezza della matrice *pointok*.
La variabile *z* rappresenta invece il numero di punti che si trovano in una sfera di raggio *distki* con centro dato dalle coordinate di *k*. Il valore di questa variabile è inizialmente -1. Il motivo di questa scelta è dato dal fatto che il codice misura anche la distanza di ogni punto da se stesso, il cui risultato sarà nullo e quindi minore del raggio scelto. In questo modo il valore finale di *z* relativo ad un certo *k* è esattamente il numero di punti che si trovano all'interno della sfera.

Considerando quindi che:

- *k* = riga del *pointok*, corrispondente al punto da analizzare.
  Variabile con valore iniziale = 1.

- *i* = riga del pointok, corrispondente al punto dal quale misurare la distanza.
  Variabile con valore iniziale = 1.

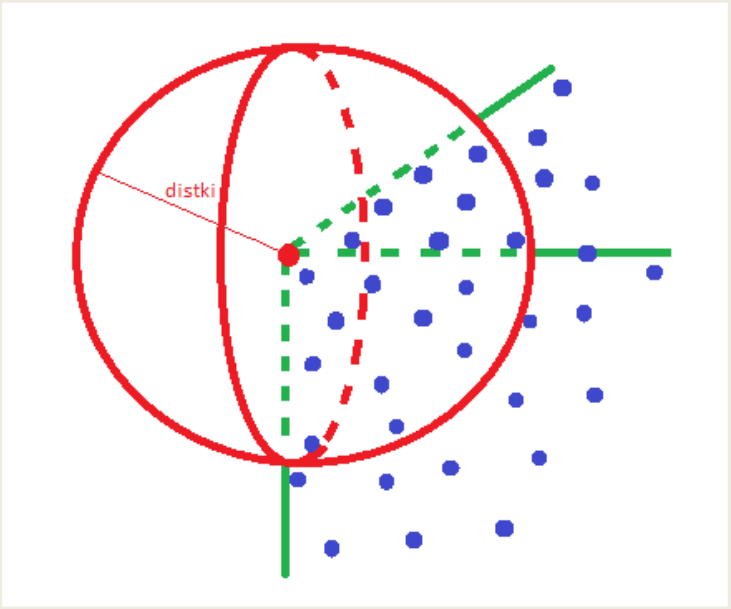
- *distki* = distanza massima per considerare i punti come vicini.
  Costante con valore scelto in base alle esigenze.

- *z* = numero di punti vicini a quello analizzato.
  Costante con valore scelto in base alle esigenze.

Il codice risulta essere il seguente:

```
k = 1;
while k <= length(pointok)
   vector.k = pointok(k,:);
   i = 1;
   z = -1;
   while i <= length(pointok)
      vector.i = pointok(i,:);
      distki = sqrt(((vector.k(1)-vector.i(1))^2)+((vector.k(2)-
     -vector.i(2))^2)+((vector.k(3)-vector.i(3))^2));
      if distki < 0.10
         z = z+1;
      end
      if z == 19
         i = length(pointok);
      end
      i = i+1;
   end
   if z < 19
      pointok(k,:) = [];
      k = k-1;
   end
   k = k+1;
end
```

Nel nostro caso è stato utilizzato un raggio *distki* = 0.10, e un valore minimo di *z* per accettare il punto pari a 19.
Ovviamente questi valori sono stati scelti in seguito a molteplici tentativi, effettuati variando la loro combinazione. Si è infatti notato che, per valori troppo piccoli di *z* ed eccessivi del raggio, resistevano alla scrematura dei piccoli gruppi di punti esterni. Questo accade perché, pur essendo questi lontani dalla nuvola principale, si ritrova all'interno di una sfera di raggio *distki* un numero di punti maggiore a *z*.
D'altra parte, incrementando troppo *z* e diminuendo il secondo, verrebbero eliminati anche i punti dell'ambiente, riconosciuti dal codice come esterni. A maggior rischio di eliminazione sarebbero quelli relativi agli angoli, o agli spigoli, perché sono circondati da punti solo per 1/8, o 1/4, della sfera attorno ad essi, come mostrato in figura:



[Fig.15] Punti a rischio eliminazione.
L'immagine chiarisce il concetto per cui un punto dell'ambiente può essere eliminato perché riconosciuto del programma come esterno. Un punto nell'angolo (punto rosso) presenta punti vicini solo in uno spicchio della sfera di raggio *distki.* Questo aumenta la possibilità che il numero dei punti vicini sia minore di z, e che il punto venga eliminato.

Lo stesso problema si avrebbe nel caso di corpi appuntiti o spigolosi. Per tale motivo valori di distki eccessivamente bassi con z elevato causerebbero l'eliminazione di punti utili, con smussamento della superficie finale.

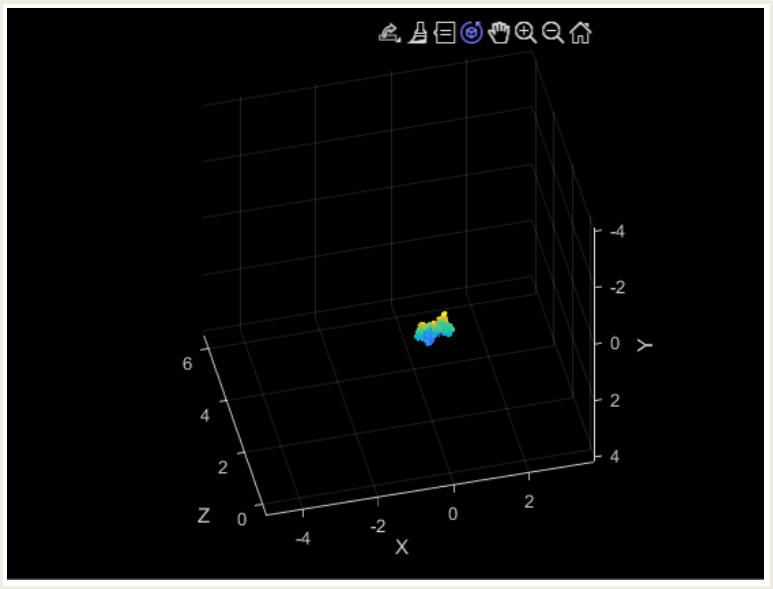
Per la ricerca dei valori ottimali sono stati analizzati più tentativi, basandosi sull'ispezione visiva di gruppi di punti distanti e sulla quantità di punti rimanenti. La combinazione scelta (distki = 0.10, z = 19) risulta essere quella che, senza presentare nuvole di punti isolate, elimina il minor numero di punti utili.

I risultati ottenuti dalle varie combinazioni sono riportati nella seguente tabella:

distki	z	punti rimanenti	nuvole isolate
0.07	6	4756	> 10
0.05	6	4671	> 10
0.04	6	4557	> 10
0.02	6	3527	7
0.07	10	4647	6
0.05	10	4533	4
0.04	10	4309	2
0.07	13	4586	1
0.05	13	4417	1
0.04	13	4193	1
0.05	16	4358	1
0.05	17	4343	1
0.07	18	4551	1
0.05	18	4326	0
0.06	18	4466	1
0.10	19	4594	0
0.07	20	4517	0
0.08	20	4563	0
0.10	20	4594	0
0.11	20	4655	3
0.12	20	4617	1
0.13	20	4610	1
0.15	20	4604	1
0.07	23	4507	0

[Tab.1] Combinazioni distki-z.
La tabella riporta i tentativi effettuati variando i valori di distki e di z e i loro risultati. Per la loro valutazione ci si è concentrati sui punti rimanenti in seguito alla scrematura (terza colonna) e il numero di nuvole di punti isolate rimanenti (quarta colonna).

Come riportato in tabella, il numero di punti rimanenti finali è 4594, ottenuto eliminando dai 5267 validi tutti i punti esterni e inevitabilmente qualche punto utile, per un totale di 673.
Il risultato finale è il seguente:



[Fig.16] Nuvola punti finale.
Il grafico riporta tutti i punti rimanenti in seguito alle scremature per validità e di punti esterni, con valori di *distki* = 0.10 e di *z* = 19. Si tratta quindi della nuvola di punti finale.

Se confrontato con la Fig.14 è possibile notare come questa nuvola appaia molto più piccola di quella precedente all'eliminazione dei punti esterni, pur avendo circa l'87% dei punti di quest'ultima. Tale aspetto fa capire come il codice principale individua punti esterni, ma che sono relativamente pochi rispetto a quelli dell'ambiente, in cui invece i punti risultano ben più compatti.

Una considerazione particolare va fatta per i casi in cui rimangono poche nuvole di punti isolate. Si può infatti pensare di eliminarle successivamente. L'idea è quella di identificare la posizione del gruppo di punti, determinare un cubo con baricentro nella

coordinata trovata, ed eliminare tutto ciò che è contenuto al suo interno.
Abbiamo dunque ideato il seguente codice:

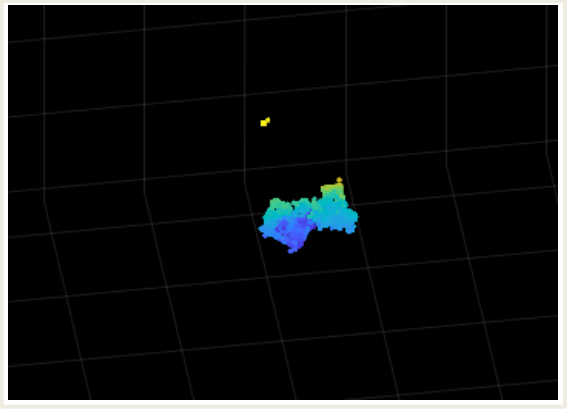
```
dist = 0.5;
xp = -0.57;
yp = -1.12;
zp = 1.25;
xpMax = xp+dist;
xpMin = xp-dist;
ypMax = yp+dist;
ypMin = yp-dist;
zpMax = zp+dist;
zpMin = zp-dist;

variab = 1
while variab < length(pointok)
   if (pointok(variab, 1)< xpMax) && (pointok(variab, 1)> xpMin) &&
      (pointok(variab, 2)< ypMax) && (pointok(variab, 2)> ypMin) &&
      (pointok(variab, 3)< zpMax) && (pointok(variab, 3)> zpMin)
      pointok(variab, :) = [];
   else
      variab = variab + 1;
   end
end
```

in cui:

- xp = coordinata x del baricentro scelto
- yp = coordinata y del baricentro scelto
- zp = coordinata z del baricentro scelto
- dist = metà del lato del cubo

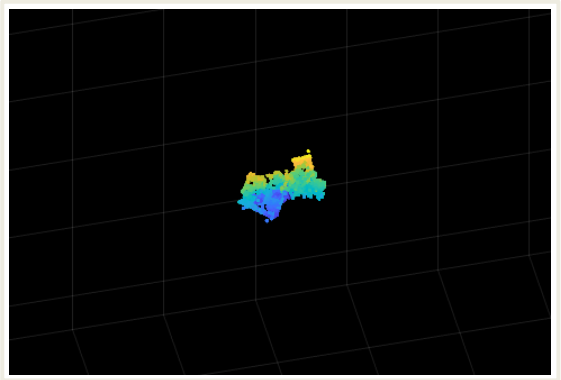
Il codice deve essere usato per una singola nuvola. Quindi nel caso di molteplici gruppi esterni è necessario effettuare il running più volte, variando a dovere i quattro valori precedentemente descritti.

Esaminando il caso in cui *distki* = 0.12 e *z* = 20, si può notare dalla tab.1 che rimangono 4617 punti con la presenza di una nuvola esterna, come mostrato in figura:



[Fig.17] Nuvola punti con gruppo esterno.
Il grafico rappresenta la nuvola di punti relativa ai valori *distki* = 0.12 e *z* = 20. Risulta ben visibile il gruppo di punti esterni che hanno evitato l'eliminazione.

Prendendo dal grafico le coordinate del gruppo di punti esterni (-0.57, -1.12, 1.25) e individuando un opportuno lato per il cubo (1) è possibile applicare il codice per eliminare la nuvola:



[Fig.18] Nuvola punti finale 2.
È riportata la stessa nuvola della Fig.17 in seguito all'eliminazione del gruppo di punti esterni.

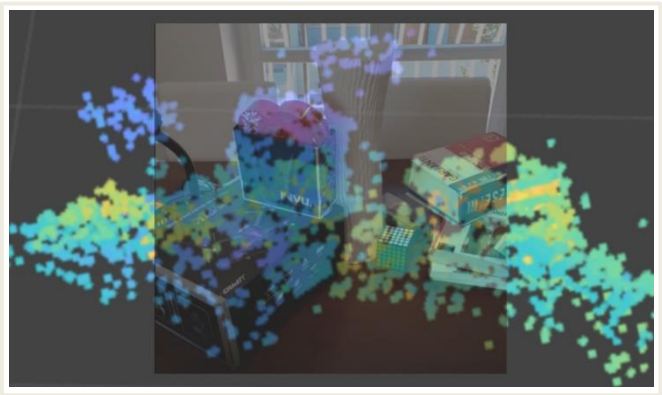
Il risultato ottenuto è una nuvola di soli punti utili, per un totale di 4586. La nuvola isolata conteneva quindi 31 punti, che spiega perché non fosse stata eliminata precedentemente (z=20).

È un procedimento tutt'altro che automatico, che richiede un certo numero di tentativi e delle procedure manuali. Per tale motivo questo approccio può essere utilizzato solo in caso di necessità di un'analisi approfondita e precisa.

4.2 Calcolo misure

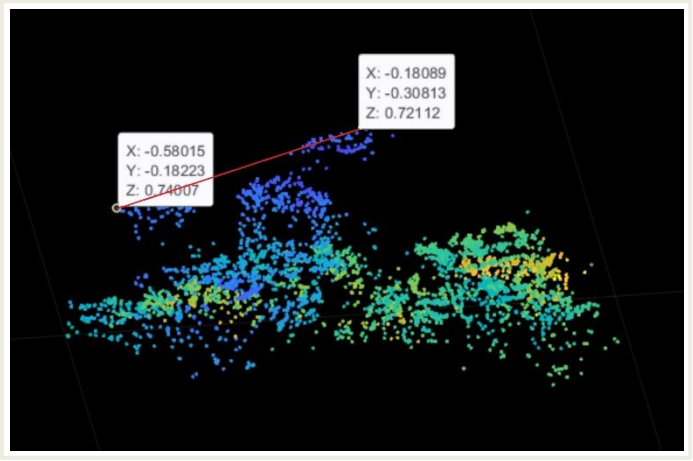
Uno degli obbiettivi finali dell'intero processo è il calcolo delle misure come, ad esempio, la distanza tra due oggetti. Il video analizzato non risulta adatto a queste richieste perché la ridotta grandezza dei soggetti e dell'intero ambiente rende la nuvola di difficile interpretazione visiva e gli errori non trascurabili.

Per il calcolo delle misure il primo step è comprendere l'orientazione della nuvola e le corrispondenze tra questa e gli oggetti reali.



[Fig.19] Orientazione nuvola.
La sovrapposizione tra la nuvola dei punti e l'immagine estrapolata dal video aiuta a comprenderne l'orientazione.

Per misurare poi la distanza tra due superfici si individua il punto più rappresentativo di ognuna e le sue coordinate.



[Fig.20] Distanza tra punti.
Sono stati selezionati due punti con le rispettive coordinate per poterne calcolare la distanza reciproca. Se confrontata con la Fig.19 è possibile notare che i punti appartengono ad un'estremità della lampada (a sinistra) e ad un punto superiore del vaso (in alto).

Si applica poi una semplice formula, trascrivibile anche su MATLAB:

```
xa = -0.58015;
ya = 0.18223;
za = 0.74007;
xb = -0,18089;
yb = -0.30813;
zb = 0.72112;
distab = sqrt(((xa-xb)^2)+((ya-yb)^2)+((za-zb)^2))
```

in cui (xa,ya,za) e (xb,yb,zb) sono le coordinate dei punti a e b selezionati.
In questo caso la distanza misurata è di 1.7321, ma è certamente un risultato poco affidabile. Una maggiore attendibilità si avrebbe nel caso di ambienti esterni, in cui l'errore relativo risulterebbe minore.

Capitolo 5

Creazione superficie

Come già anticipato nei precedenti capitoli, la tecnologia SLAM viene utilizzata anche per la riproduzione grafica dell'ambiente che si sta analizzando. Dopo aver estratto e ripulito la nuvola di punti può essere quindi utile utilizzarla per creare le superfici.

Un codice utile [8] per tale scopo è il seguente:

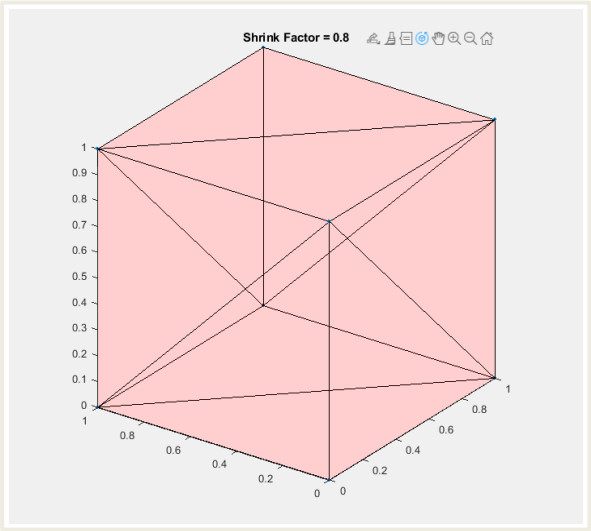
```
pointDouble=double(pointok);
k = boundary(pointDouble,.8);
figure
plot3(pointDouble(:,1),pointDouble(:,2),pointDouble(:,3),'.'…
        ,'MarkerSize',10)
hold on
trisurf(k,pointDouble(:,1),pointDouble(:,2),pointDouble(:,3)…
        ,'FaceColor','red','FaceAlpha',0.1)
axis equal
title('Shrink Factor = 0.8')
```

Dopo aver raddoppiato la precisione dei punti contenuti nella matrice *pointok* tramite *double*, la funzione *boundary* restituisce un confine conforme tra i questi. In particolare, il processo viene effettuato tramite triangolazione tra i punti.
La matrice *k* presenta tre colonne, in cui sono riportati i punti che fungono da vertice di ogni triangolo, e tante colonne quanti i triangoli utilizzati per la creazione della superficie.

Per chiarire questo concetto è possibile semplificare la nuvola iniziale, utilizzando, ad esempio, 8 punti disposti in forma cubica di lato 1. La matrice dei punti su cui lavoriamo può essere quindi:

```
pointok = [0,0,0; 1,0,0; 0,1,0; 0,0,1; 1,1,0; 1,0,1; 0,1,1; 1,1,1]
```

Applicando ora il codice al nuovo *pointok*, otteniamo questo risultato:



[Fig.21] Superficie cubica.
Applicando il codice agli 8 punti del *pointok* si forma un cubo, i cui vertici sono proprio i punti di partenza.

Il cubo che ci aspettavamo di ricavare è ben visibile, così come i triangoli utilizzati per crearlo. In particolare, il sistema dispone due triangoli isosceli su ogni faccia per formare il quadrato.

A conferma di questo è possibile analizzare la matrice k, che sarà composta da 12 righe, una per ogni triangolo:

Variables - k

k

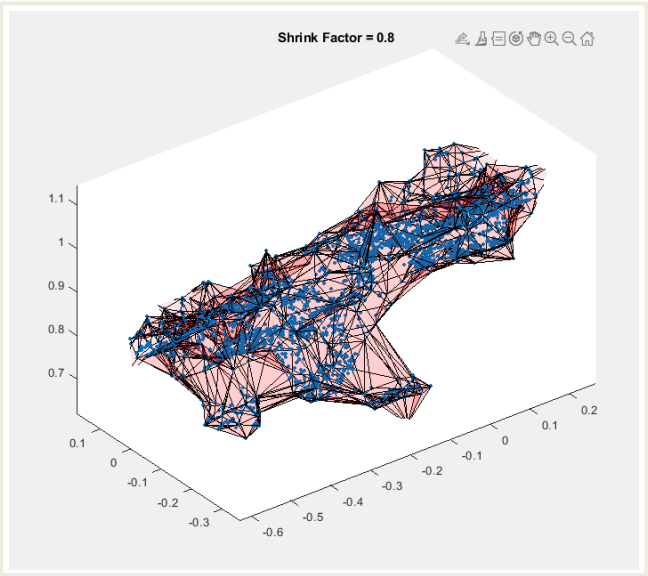
12x3 double

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

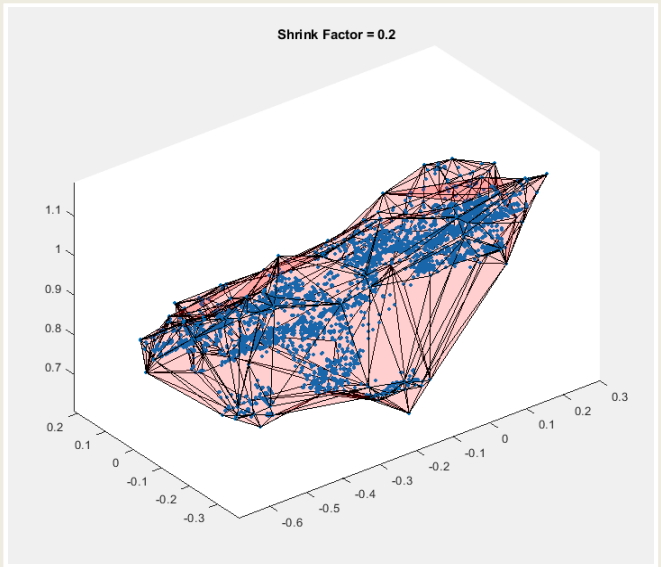
[Fig.22] Matrice k.
La matrice presenta sulle righe i triangoli utilizzati per la creazione della superficie, e sulle colonne i punti che fungono da vertici di ogni triangolo.

Un altro aspetto importante del codice è il cosiddetto *shrink factor*, cioè il fattore di restringimento. Si tratta [9] di uno scalare compreso tra 0 e 1 che determina la contrazione della superficie, ed è il secondo input della funzione *boundary*.
In particolare, per uno *shrink factor* pari a 0 si ottiene una superficie convessa. Avvicinandosi sempre più al valore unitario questa avvolge i punti risultando più contratta.

Di seguito sono riportati due risultati grafici ottenuti per diversi fattori di restringimento:



[Fig.23] Superficie Shrink Factor 0.8.
Il fattore di contrazione vicino a 1 determina una superficie più contratta e un maggior numero di triangoli (1520) per la sua creazione.



[Fig.23] Superficie Shrink Factor 0.2.
Il fattore di contrazione vicino a 0 determina una superficie più convessa e un minor numero di triangoli (470) per la sua creazione.

Capitolo 6

Conclusione

Quanto trattato in questa tesi cerca di aggiungere nuovi tasselli all'innovativa tecnologia SLAM, che è in costante fase di sperimentazione e miglioramento.

Dopo aver compreso i meccanismi generali con cui questa opera, è stato possibile affrontare nuove problematiche derivanti dal passaggio dalla lettura di frames sequenziale a quella diretta di file video. L'analisi, di carattere generale, si è incentrata particolarmente su un esempio concreto, per il quale sono stati impiegati mezzi e dispositivi accessibili a chiunque. La semplicità di utilizzo, dietro cui si celano concetti complessi, è l'ennesimo punto di forza di questa tecnologia, per la quale si prospetta un futuro di grande impego in diversi ambiti quotidiani.

I risultati derivanti da questo studio possono sembrare limitati, ma i codici e i dati che esso espone sono potenzialmente trattabili in altre applicazioni mirate a contribuire al progresso della tecnologia SLAM.

Bibliografia

[1] Nour Naghi, "Localization and mapping in dynamic physical systems", in *Simultaneous Localization And Mapping Technologies,* 2018, pp. 17-23. Available: https://amslaurea.unibo.it/17852/1/Tesi-Naghi.pdf

[2] Mur-Artal, Raul, Jose Maria Martinez Montiel, and Juan D. Tardos. "ORB-SLAM: a versatile and accurate monocular SLAM system", in *IEEE Transactions on Robotics* 31, no. 5, 2015, pp. 1-8. Available: https://zaguan.unizar.es/record/32799/files/texto_completo.pdf

[3] "Monocular Visual Simultaneous Localization and Mapping", in *MathWorks, Help Center,* n.d. Available: https://it.mathworks.com/help/vision/examples/monocular-visual-simultaneous-localization-andmapping.html#MonocularVisualSimultaneousLocalizationAndMappingExample-12

[4] "Focal Length", *Wikipedia*, 2020. Available: https://en.wikipedia.org/wiki/Focal_length

[5] "Input Arguments", in *CameraIntrinsics, MathWorks, Help Center,* n.d. Available: https://it.mathworks.com/help/vision/ref/cameraintrinsics.html#d120e55621

[6] "Principal planes and points", in *Cardinal point (optics), Wikipedia,* 2020. Available: https://en.wikipedia.org/wiki/Cardinal_point_(optics)#Principal_planes_and_points

[7] "helperMapPointSet.m", in *MATLAB,* n.d. Available:

[8] "Boundary", in *MathWorks, Documentation,* n.d. Available:

[9] "Shrink Factor Effect on 3-D Boundary", in *Boundary, MathWorks, Help Center,* n.d. Available: https://it.mathworks.com/help/matlab/ref/boundary.html