POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea Magistrale

Misura di parametri di dinamica del veicolo mediante smartphone

Relatore
prof. Stefano Mauro

Correlatori:
prof. Antonio Servetti
prof. Enrico Galvagno

Laureando
Marco Di Rosa
matricola: 233075

Anno accademico 2017 – 2018

Alla mia preziosa famiglia, agli amici veri, ai miei coinquilini Peppe, Gianni, Alessio, a Martina e a tutte le persone che mi sono state vicine.

Sommario

Il lavoro portato a termine in questi sei mesi è incentrato sul calcolo dei parametri di dinamica di un veicolo con particolare attenzione per l'angolo di assetto dello stesso, utilizzando come sistema di acquisizione ed elaborazione dati un comune smartphone, su cui verrà installata un'applicazione ad hoc per l'elaborazione dei dati rilevati tramite i sensori dello smartphone. Non si prevede dunque di poter accedere ad altri strumenti come fatto il precedenza (rete CAN del veicolo, piattaforma inerziale esterna al veicolo (IMU)).

Nei primi capitoli della tesi si illustreranno alcuni argomenti di carattere teorico sul calcolo dell'angolo di assetto di un veicolo, sulle misure che entrano in gioco nel suddetto calcolo, dell'approccio scelto per andare in contro a meno errori possibili, ed una breve introduzione alla piattaforma di sviluppo, sul linguaggio di programmazione utilizzato e sulla sensoristica del terminale.

Successivamente verranno trattati in dettaglio gli algoritmi sviluppati per il calcolo dei parametri di dinamica, del flusso di elaborazione, sulla memorizzazione interna dei dati e sulla rappresentazione finale degli stessi. Si presterà forte attenzione sull'algoritmo sviluppato per il calcolo dell'angolo volante, andando ad analizzare l'approccio iniziale, le scelte fatte ed i possibili miglioramenti di quest'ultimo.

Nella fase finale verranno analizzate le prove su strada effettuate, atte a validare il metodo di elaborazione ed acquisizione proposto, verranno confrontati i risultati ottenuti con quelli elaborati mediante un altro sistema di elaborazione (MatLab), e verranno proposti alcuni miglioramenti ed analizzate alcune fasi critiche dello sviluppo della piattaforma.

Ringraziamenti

Un ringraziamento speciale va ai miei relatori: prof. ***Stefano Mauro***, prof. ***Enrico Galvagno***, prof. ***Antonio Servetti***, per avermi aiutato in molte situazioni a tirar fuori il meglio di me e motivandomi dinanzi ad un'infinità di problemi avuti durante il lavoro svolto; per aver avuto pazienza in molte occasioni in cui ho dovuto impiegare la maggior parte del tempo per capire e comprendere argomenti *aim*è sconosciuti fino a poco tempo fa. Un ringraziamento particolare anche al prof. ***Stefano Pastorelli***, per avermi aiutato con la stesura del codice relativo alla calibrazione dello smartphone nel sistema di riferimento veicolo, veramente molto disponibile e chiaro nelle spiegazioni. Un ringraziamento anche al mio collega ing. ***Roberto Marturano***, per avermi affiancato durante il lavoro iniziale di conversione e implementazione degli algoritmi da MatLab ad Android e per avermi dato preziosi consigli per le implementazioni future. Ovviamente ringrazio anche tutta la mia *famiglia*, per la pazienza avuta con me durante questo lungo periodo di stesura della tesi; i miei *coinquilini* e gli *amici* più fidati, per avermi supportato durante questi mesi. L'ultimo ringraziamento va ad una persona speciale, ***Martina***, che mi è stata vicino durante tutto il periodo di lavoro, a volte anche dandomi preziosi consigli, anche se non esperta in materia, ma che col cuore è riuscita sempre e comunque a starmi vicino e non farmi mai mollare.

Indice

Elenco delle figure

Parte I

Prima Parte

Capitolo 1

Introduzione

1.1 Principi generali e grandezze in gioco

L'obiettivo del lavoro svolto è quello di riuscire a stimare l'angolo di assetto β, il quale rappresenta l'angolo fra la direzione longitudinale del moto e la direzione della velocità V, può essere valutato in qualunque punto del veicolo, nel seguito lo si valuterà in corrispondenza del baricentro del veicolo stesso. Nel momento in cui un veicolo sterza o, in situazioni di emergenza, deve essere corretta la traiettoria dello stesso, una grandezza che entra in gioco nei sistemi di controllo di stabilità del veicolo è proprio l'angolo di assetto, il quale fornisce informazioni della stabilità del veicolo durante tali manovre. La misura diretta del β richiede comunque l'utilizzo di dispositivo costosi e solitamente non disponibili nelle vetture di utilizzo standard, così questo viene stimato tramite l'utilizzo di altre grandezze che vedremo in seguito. Una stima precisa del β può contribuire a migliorare di molto la stabilità e la sicurezza di guida.

Gli algoritmi utilizzati per il calcolo del β sono stati sviluppati precedentemente presso il dipartimento di meccanica del Politecnico di Torino [1], per poi essere stati applicati utilizzando una piattaforma inerziale [2].

Oltre al suddetto β vi sono altre grandezze che entrano in gioco nel calcolo dei parametri di dinamica di un veicolo:

- δ_v: angolo sterzo
- u_n: velocità longitudinale
- a_x: accelerazione longitudinale
- a_y: accelerazione laterale
- a_z: accelerazione verticale
- $\dot{\psi}$: velocità di imbardata

Per ottenere una buona stima delle grandezze sopra elencate utilizzeremo dei sensori presenti in ogni smartphone moderno:

- Accelerometro
- Giroscopio
- GPS
- Fotocamera

Rispettivamente con accelerometro e giroscopio è possibile calcolare le componenti di accelerazione e velocità angolare, tramite il gps otteniamo invece una stima della traiettoria del veicolo e la sua velocità istantanea, con la fotocamera invece si riesce a dare una stima dell'angolo sterzo, puntando l'obiettivo verso quest'ultimo ed elaborando l'immagine affinché vengano riconosciuti, ad ogni frame, dei markers montati proprio sullo sterzo.

1.2 Modelli di veicolo

Per riuscire a stimare correttamente i parametri in gioco bisogna applicare i calcoli ad un modello di veicolo che non potrà mai riflettere quello reale, per cui è stato posto un appunto sulla discretizzazione del modello: partendo dapprima da uno piuttosto semplificato (Modello monotraccia) per poi andare man mano a complicarlo inserendo sempre più parametri di ingresso su cui lavorare.

1.2.1 Accenno ai sistemi dinamici non lineari

Per poter applicare, come vedremo in seguito, i filtri di Kalman lineari ai nostri calcoli, è necessario che il modello proposto per rappresentare la dinamica di una autovettura rispecchi un modello lineare; purtroppo ciò non è sempre facile da attuare poiché la maggior parte dei modelli presenti nella realtà ha un comportamento del tutto diverso da una relazione lineare tra ingresso e uscita. Per poter linearizzare in maniera corretta un sistema bisogna semplificare un modello non lineare, scegliendo quali *stati* includere nel sistema. Bisogna anche discretizzare i *parametri* che rappresentano un sistema non lineare, per poter in qualche modo semplificare i calcoli. Tale stima viene effettuata a seconda della tipologia di modello applicando dei valori in ingresso al sistema reale e confrontare la risposta con quella del modello semplificato. Infine bisogna testare se il modello stimato rappresenta un buon modello applicando un diverso set di dati rispetto a quello scelto in fase di stima.

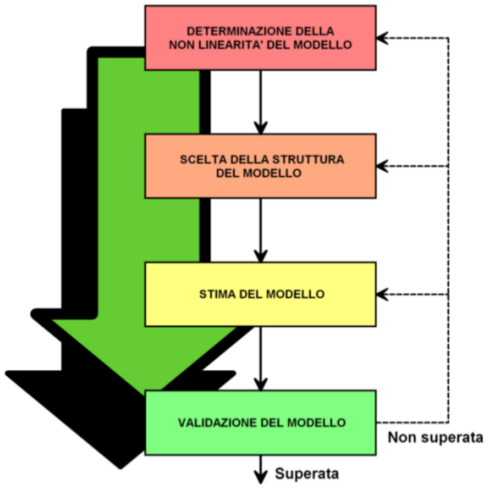


Figura 1.1. Tipico flusso per determinare un modello lineare.

1.2.2 Modello di veicolo monotraccia lineare

Partendo dai presupposti sopraelencati è possibile risalire a un modello molto semplificato di autoveicolo a 3 G.d.l Tale modello ha due stati:

- v: velocità laterale
- $\dot{\psi}$: velocità di imbardata, ossia velocità di rotazione del veicolo attorno ad un asse verticale

E un ingresso:

- δ_v: angolo di rotazione del volante

Come si può facilmente notare manca una grandezza fondamentale che prima avevamo considerato ossia u_n, velocità longitudinale che in questo modello viene considerata costante.

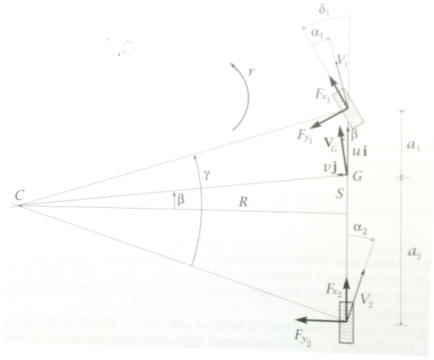


Figura 1.2. Struttura del modello monotraccia.

Tramite calcoli effettuati precedentemente[1] si riesce ad ottenere un set di equazioni che descrivono lo stato del sistema, dette equazioni del *moto* così rappresentate:

$$\dot{v} = V_2 v + R_2 \dot{\psi} + D_2 \delta_v \tag{1.1}$$

$$\ddot{\psi} = V_3 v + R_3 \dot{\psi} + D_3 \delta_v \tag{1.2}$$

con:

- $V_2 = -\frac{C_1+C_2}{mu}$
- $R_2 = -(\frac{C_1 a_1 - C_2 a_2}{mu} + u)$
- $D_2 = \frac{C_1 \tau}{m}$
- $V_3 = -\frac{C_1 a_1 - C_2 a_2}{Ju}$
- $R_3 = -(\frac{C_1 a_1^2 + C_2 a_2^2}{Ju} + u)$
- $D_3 = \frac{C_1 a_1}{J} \tau$

Nello spazio degli stati tali equazioni possono essere reinterpretate sotto forma di matrice, sapendo che un sistema dinamico qualunque può essere descritto come:

$$\dot{x} = Ax + Bu \tag{1.3}$$

Nel nostro caso otteniamo lo spazio degli stati rappresentato come:

$$\begin{pmatrix} \dot{v} \\ \ddot{\psi} \end{pmatrix} = Ax + Bu = \begin{bmatrix} V_2 & R_2 \\ V_3 & R_3 \end{bmatrix} \begin{pmatrix} v \\ \dot{\psi} \end{pmatrix} + \begin{bmatrix} D_2 & 0 \\ D_3 & 0 \end{bmatrix} \begin{pmatrix} \delta_v \\ 0 \end{pmatrix} \tag{1.4}$$

E' possibile allora poter simulare gli andamenti di v e $\dot{\psi}$ modificando i parametri visti precedentemente ed ottenere una risposta dal sistema del tipo:

$$y = Cx + Du = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{pmatrix} v \\ \dot{\psi} \end{pmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{pmatrix} \delta_v \\ 0 \end{pmatrix} \tag{1.5}$$

Come vedremo successivamente, andando a modificare i seguenti parametri:

- $C1$, rigidezza deriva dei pneumatici anteriori
- $C2$, rigidezza deriva dei pneumatici posteriori
- $a1$, semipasso anteriore
- $a2$, semipasso posteriore
- τ, rapporto sterzo
- J, momento di inerzia del veicolo rispetto ad un asse verticale del baricentro
- m, massa del veicolo

si può predisporre un certo tipo di autovettura al test, direttamente preimpostando i modelli noti direttamente all'interno del codice dell'applicazione.

1.2.3 Estrapolazione del modello Cinematico di autovettura

Considerando il veicolo come un corpo rigido che si muove su un piano xy è possibile ricavare la seguente relazione che descrive il legame tra le due componenti di accelerazione del baricentro del veicolo (longitudinale a_x e laterale a_y) e le componenti di velocità rispetto al sistema di riferimento solidale al veicolo. Il modello cinematico di autovettura, basato sui parametri appena descritti:

$$\begin{cases} a_x = \dot{u} - \dot{\psi} v \\ a_y = \dot{v} + \dot{\psi} u \end{cases} \tag{1.6}$$

con:

- a_x, accelerazione longitudinale
- a_y, accelerazione laterale
- $\dot{\psi}$, velocità di imbardata
- u, velocità longitudinale
- v, velocità laterale

Tali equazioni possono essere scritte in forma di matrice come:

$$\boxed{\begin{pmatrix} \dot{u} \\ \dot{v} \end{pmatrix} = \begin{bmatrix} 0 & \dot{\psi} \\ -\dot{\psi} & 0 \end{bmatrix} \begin{pmatrix} u \\ v \end{pmatrix} + \begin{pmatrix} a_x \\ a_y \end{pmatrix}} \tag{1.7}$$

Tale relazione rappresenta gli stati del modello cinematico di autovettura.

1.2.4 Estrapolazione del modello Dinamico di autovettura

Allora stesso modo [1], il modello dinamico viene rappresentato con le due equazioni di equilibrio:

$$\dot{v} = V_2 v + R_2 \dot{\psi} + U_2 u + D_2 \delta_v + K_2 \tag{1.8}$$

$$\ddot{\psi} = V_3 v + R_3 \dot{\psi} + U_3 u + D_3 \delta_v + K_3 \tag{1.9}$$

Definito il vettore degli stati come:

$$x = \begin{pmatrix} v \\ \dot{\psi} \end{pmatrix} \tag{1.10}$$

E quello delle entrate come:

$$u = \begin{pmatrix} u \\ \delta_v \end{pmatrix} \tag{1.11}$$

Otteniamo la scrittura:

$$\boxed{\begin{pmatrix} \dot{v} \\ \ddot{\psi} \end{pmatrix} = \begin{bmatrix} V_2 & R_2 \\ V_3 & R_3 \end{bmatrix} \begin{pmatrix} v \\ \dot{\psi} \end{pmatrix} + \begin{bmatrix} U_2 & D_2 \\ U_3 & D_3 \end{bmatrix} \begin{pmatrix} u \\ \delta_v \end{pmatrix} + \begin{pmatrix} K_2 \\ K_3 \end{pmatrix}} \tag{1.12}$$

Rappresenta lo spazio degli stati del modello dinamico di autovettura.

Capitolo 2

Discretizzazione dei modelli dinamici

Per poter implementare i modelli occorre passare da un sistema *continuo* ad uno *discreto*. Per far ciò occore dividere il dominio del tempo in piccoli intervalli, detti campioni, rappresentati ognuno uno stato del sistema in quell'istante.

Dalla formula di *Eulero* sappiamo che uno stato generico x di un sistema può essere rappresentato come:

$$\dot{x} = \frac{dx}{dt} \simeq \frac{x_{k+1} - x_k}{T_s} \tag{2.1}$$

Con T_s tempo di campionamento, x_{k+1} lo stato del sistema all'istante $k+1$, x_k lo stato del sistema all'istante k.

2.1 Discretizzazione del modello cinematico

Come precedentemente descritto (1.2.3) il modello cinematico può essere descritto dalla (1.2.3) in cui le due equazioni di equilibrio sono:

$$\begin{aligned} a_x &= \dot{u} - \dot{\psi} v \\ a_y &= \dot{v} + \dot{\psi} u \end{aligned} \tag{2.2}$$

Evidenziando gli stati:

$$\begin{aligned} \dot{u} &= \dot{\psi} v + a_x \\ \dot{v} &= -\dot{\psi} u + a_y \end{aligned} \tag{2.3}$$

Applicando la formula di *Eulero*:

$$\begin{aligned} \frac{u_{k+1} - u_k}{T_s} &= \Psi_k v_k + a_{x_k} \\ \frac{v_{k+1} - v_k}{T_s} &= \Psi_k u_k + a_{y_k} \end{aligned} \tag{2.4}$$

Semplificando:

$$\begin{aligned} u_{k+1} &= T_s \Psi_k v_k + T_s a_{x_k} + u_k \\ v_{k+1} &= -T_s \Psi_k u_k + T_s a_{y_k} + v_k \end{aligned} \tag{2.5}$$

Scrivendo il tutto nello spazio degli stati sotto forma di matrici:

$$\begin{pmatrix} u_{k+1} \\ v_{k+1} \end{pmatrix} = \begin{bmatrix} 1 & T_s \Psi_k \\ -T_s & \Psi_k 1 \end{bmatrix} \begin{pmatrix} u_k \\ v_k \end{pmatrix} + \begin{bmatrix} T_s & 0 \\ 0 & T_s \end{bmatrix} \begin{pmatrix} a_{x_k} \\ a_{y_k} \end{pmatrix} \tag{2.6}$$

Utilizzando la scrittura generica si ottiene:

$$\boxed{x_{k+1} = A_d x_k + T_s B u_k + T_s K} \tag{2.7}$$

2.2 Discretizzazione del modello dinamico

Come visto, applicando la formula di *Eulero* ad un'equazione nel continuo è possibile ottenere la funzione discretizzata in certi istanti di tempo. Allo stesso modo del modello cinematico, in quello dinamico, partendo dalla (1.2.4) con le due equazioni di equilibrio:

$$\begin{aligned} \dot{v} &= V_2 v + R_2 \dot{\psi} + U_2 u + D_2 \delta_v + K_2 \\ \ddot{\psi} &= V_3 v + R_3 \dot{\psi} + U_3 u + D_3 \delta_v + K_3 \end{aligned} \tag{2.8}$$

Applicando la formula di *Eulero*:

$$\begin{aligned} \frac{v_{k+1} - v_k}{T_s} &= V_2 v_k + R_2 \Psi_k + U_2 u_k + D_2 \delta_{v_k} + K_2 \\ \frac{\Psi_{k+1} - \Psi_k}{T_s} &= V_3 v_k + R_3 \Psi_k + U_3 u_k + D_3 \delta_{v_k} + K_3 \end{aligned} \tag{2.9}$$

Semplificando:

$$\begin{aligned} v_{k+1} &= (1 + T_s V_2) v_k + T_s R_2 \Psi_k + T_s U_2 u_k + T_s D_2 \delta_{v_k} + T_s K_2 \\ \Psi_{k+1} &= T_s V_3 v_k + (1 + T_s R_3) \Psi_k + T_s U_3 u_k + T_s D_3 \delta_{v_k} + T_s K_3 \end{aligned} \tag{2.10}$$

Passando allo spazio degli stati in forma matriciale:

$$\begin{pmatrix} v_{k+1} \\ \Psi_{k+1} \end{pmatrix} = \begin{bmatrix} (1 + T_s V_2) & T_s R_2 \\ T_s V_3 & (1 + T_s R_3) \end{bmatrix} \begin{pmatrix} v_k \\ \Psi_k \end{pmatrix} + \begin{bmatrix} T_s U_2 & T_s D_2 \\ T_s U_3 & T_s D_3 \end{bmatrix} \begin{pmatrix} u_k \\ \delta_{v_k} \end{pmatrix} + \begin{pmatrix} T_s K_2 \\ T_s K_3 \end{pmatrix} \tag{2.11}$$

Utilizzando la scrittura generica:

$$\boxed{x_{k+1} = A_d x_k + T_s B u_k + T_s K} \tag{2.12}$$

Capitolo 3

Il Filtro di Kalman lineare

Il filtro di Kalman è un filtro che opera in maniera ricorsiva, valutando lo stato del sistema dinamico. Parte da una serie di misurazioni soggette a rumore, che nel nostro caso saranno i dati grezzi ottenuti dai sensori, e facendo riferimento ad un *modello* dinamico riduce al massimo il rumore sui dati in ingresso. Il filtro fornisce allo stato del sistema due contributi significativi: il contributo *predittivo* che si basa sullo stato precedente del sistema,opera sulla stima dell'errore precedente e di come questo evolve nel tempo. Il contributo *correttivo* si basa invece sullo stato attuale del sistema, ed effettua un confronto tra la grandezza stimata rispetto al modello a cui fa riferimento il filtro e la stessa grandezza misurata. E' indispensabile che le due grandezze siano *coerenti*.

3.0.1 Il modello di riferimento

Come detto in precedenza un sistema dinamico può essere rappresentato mediante la (1.3), aggiungendo un'ulteriore equazione che descrive le misure in relazione allo stato del sistema:

$$y = Cx + Du \tag{3.1}$$

- C, stati disponibili
- D, ingressi disponibili

Applicando la formula di *Eulero* come in (2.7) si ottiene:

$$\begin{aligned} x_k + 1 &= A_d x_k + T_s B u_k \\ y_k &= C x_k + T_s D u_k \end{aligned} \tag{3.2}$$

Queste equazioni descrivono la parte *dinamica* del filtro di Kalman.

3.0.2 Matrici di covarianza

Oltre al modello, come precedentemente accennato, abbiamo bisogno di sviluppare la parte *reale* del filtro, per riuscire a capire quanto le misure stimate siano coerenti con il modello *ideale*. Il calcolo della parte reale del filtro di Kalman avviene calcolando tre matrici di covarianza:

- Q, matrice di covarianza che ci da informazioni su quanto la stima del modello sia affidabile rispetto alla misura, bisogna osservare i valori sulla diagonale; più basso è il valore sulla diagonale più affidabile sarà il modello.

$$Q = \begin{bmatrix} w_{k1} & \cdots & 0 \\ \vdots & \ddots & 0 \\ 0 & \cdots & w_{kn} \end{bmatrix} \tag{3.3}$$

  w_k rappresenta una stima dell'errore associato al modello.

- R, matrice di covarianza che ci da informazioni su quanto la misura sia affidabile rispetto al modello teorico, osservando i valori sulla diagonale; più è basso il valore più la misura sarà affidabile.

$$R = \begin{bmatrix} z_{k1} & \cdots & 0 \\ \vdots & \ddots & 0 \\ 0 & \cdots & z_{kn} \end{bmatrix} \tag{3.4}$$

  z_k rappresenta una stima dell'errore associato alla misura.

- P, matrice di covarianza di stima del filtro; viene calcolata in funzione di Q (3.3), R (3.4) e del *guadagno di Kalman* (3.0.3). La matrice P è aggiornata ad ogni iterazione del filtro, possiamo quindi definire come P_k^- la matrice di covarianza del filtro calcolata *a priori* prima di procedere alla correzione allo step k. In questo modo è possibile definire una condizione iniziale del sistema (all'inizio del ciclo):

$$P_k^- = A_{dk} P_{k-1} A_{dk}^T + Q \tag{3.5}$$

  Con:

  - A_{dk}, matrice del modelo dinamico
  - P_{k-1}, matrice di covarianza del filtro calcolata all'iterazione precedente (definita dall'equazione di Ricciati [4])

Avendo calcolato le tre matrici appena descritte e con la 3.2, è possibile implementare l'algoritmo.

3.0.3 Guadagno di Kalman

Come precedentemente accennato il guadagno di Kalman è necessario per implementare lo stesso algoritmo alle stime; rappresenta un vettore in funzione dei parametri appena descritti, delle misure grezze effettuate e del sistema allo stato k. Viene calcolato una volta ad ogni iterazione, e nel caso di modello lineare viene rappresentato come:

$$K_k = P_k^- C^T (C P_k^- C^T + R)^{-1} \tag{3.6}$$

3.0.4 Ciclo di funzionamento (Kalman lineare)

Avendo descritto esaustivamente il funzionamento e l'inizializzazione del filtro di Kalman, adesso porremo l'appunto su come vengono definite le fasi del ciclo di funzionamento dell'algoritmo, partendo dall'*inizializzazione*:

- Inizializzazione dello stato, $x_k(k=1)$ contiene gli stati misurati, quindi avremo tante righe quanti sono il numero di stati misurati (n).

$$x_k(k=1) = \begin{pmatrix} x_{1_m} \\ \vdots \\ x_{n_m} \end{pmatrix} \tag{3.7}$$

- Inizializzazione della matrice di covarianza P (3.0.2), questa viene posta $= Q$ come prima ipotesi.

$$P^-(k=1) = Q \tag{3.8}$$

Terminata la fase preliminare di inizializzazione, si definisce lo stato del sistema (3.2) a priori come:

$$x_k^- = A_{dk}x_{k-1} + T_sBu_{k-1} \tag{3.9}$$

con k stato del sistema corrente, $k-1$ stato del sistema allo step precedente.

Successivamente si calcola il primo valore della matrice P, a priori:

$$P_k^- = A_{dk}P_{k-1}A_{dk}^T + Q \tag{3.10}$$

Avendo calcolato P si può procedere con il calcolo del guadagno di Kalman (3.0.3), che servirà a correggere il sistema.

Ottenuti tutti i parametri all'istante di campionamento k si procede ad aggiornare lo stato:

$$x_k = x_k^- + K_k(y_k - Cx_k^-) \tag{3.11}$$

con:

$$y_k = Cx_k, \tag{3.12}$$

misura all'iterazione k-esima. Infine, utilizzando il guadagno di Kalman riaggiorniamo la matrice P:

$$P_k = [I - K_kC]P_k^-[I - K_kC]^T + K_kR_kK_k^T \tag{3.13}$$

con I matrice identità.

Tale ciclo viene ripetuto fino ad ottenere un vettore degli stati stimato.

3.1 Applicazioni

3.1.1 Applicazione del filtro di Kalman al modello cinematico

Il filtro appena descritto può essere applicato a qualunque sottosistema lineare, in questo paragrafo andremo a vedere come può essere applicato alla parte *cinematica* del modello di autovettura descritto (1.2.3), senza considerare quindi la parte dinamica dello stesso. Il tutto può essere schematizzato come segue:

- **Ingressi:**
  - a_x, accelerazione longitudinale
  - a_y, accelerazione laterale
- **Stati:**
  - u, velocità longitudinale
  - v, velocità laterale
- **Misure:**
  - a_x, accelerazione longitudinale
  - a_y, accelerazione laterale
  - $\dot{\psi}$, velocità imbardata
  - u, velocità longitudinale

Come verificato da calcoli sperimentali [1], l'angolo d'assetto β è ricavato risolvendo la 1.2.3 e calcolando u e v e da esse l'angolo:

$$\beta = atan(\frac{v}{u}) \tag{3.14}$$

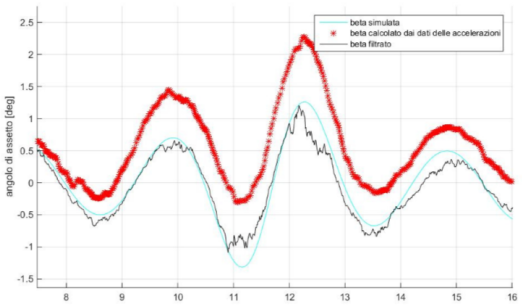


Figura 3.1. Confronto dei β nel caso di modello cinematico.

3.1.2 Applicazione del filtro di Kalman al modello dinamico

Allo stesso modo in questo paragrafo analizzeremo la parte *dinamica* del modello di autovettura (1.2.4). Per comodità si riporano i parametri:

- **Ingressi:**
  - u, velocità longitudinale
  - $delta_v$, angolo volante
- **Stati:**

  - v, velocità laterale
  - $\dot{\psi}$, velocità di imbardata

- **Misure:**
  - $delta_v$, angolo volante
  - $\dot{\psi}$, velocità imbardata
  - u, velocità longitudinale

Anche qui, come verificato [1], risolvendo l'equazione differenziale 1.2.4 e ricavando u e v, otteniamo:

$$\beta = atan(\frac{v}{u}) \tag{3.15}$$

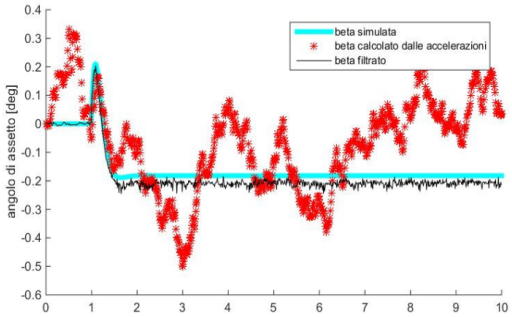


Figura 3.2. Confronto dei β nel caso di modello dinamico.

Capitolo 4

Il problema dell'angolo volante

La stima del parametro δ_v, angolo volante, è risultata essere più problematica di quanto si pensasse inizialmente poiché nei lavori di tesi precedenti [2, 3] il calcolo era svolto in *post*, quindi veniva registrato un video del volante e successivamente elaborato su Matlab per estrapolare le varie posizioni dei marker e di conseguenza l'angolo volante ad un certo istante di tempo. Tutto ciò è adesso risultato molto più complesso poichè il calcolo dell'angolo avviene a *runtime*; l'algoritmo è ovviamente diverso rispetto a quello ideato inizialmente su Matlab ma si rifà, come ragionamento base, a riuscir ad individuare frame dopo frame la posizione di tali *markers*, dei semplici cerchi bianchi su fondo nero, posizionati sullo sterzo del veicolo da esaminare.

Figura 4.1. Esempio di applicazione di markers su un volante generico

Inizialmente l'intento era quello di replicare l'algoritmo sviluppato su Matlab ed adattarlo alla piattaforma Android. Volendo però riuscire ad eseguire il calcolo a runtime l'algoritmo doveva essere pesantemente rivisto; così si è optato per uno sviluppo *da zero*

di quest'ultimo. Una delle prime fasi di sviluppo ha visto l'implementazione tramite un algoritmo noto, conosciuto come $Lukas - Kanade\ method$ [5], un algoritmo specializzato per il calcolo dell'$optical\ flow$: in poche parole seguiva la logica secondo cui il *neighborhood*(letteralmente il "vicinato", un generico gruppo di pixel adiacenti ad un pixel di riferimento) di un generico pixel, considerato come il "vicinato locale", quindi una zona molto ristretta in termini di pixel, in una generica zona dell'immagine si mantiene costante; nel momento in cui si presenta uno spostamento nell'immagine tali pixel si "muovono" tutti con la stessa velocità. L'algoritmo generico utilizza una $finestra$ di calcolo che man mano viene spostata nell'intera immagine per riuscire ad interpretare dei movimenti. Inoltre l'algoritmo è tarato per dare più *peso* ai pixel vicini al centro della finestra e meno a quelli lontani. L'approccio è molto simile al generico *metodo dei minimi quadrati*. Tale metodo era stato inizialmente scelto poiché la libreria *OpenCv*, una libreria grafica di cui si parlerà successivamente, supporta un'implementazione nativa dell'algoritmo: di fatto presa un'immagine si riesce ad estrapolare dei *punti di interesse* tali per cui l'algoritmo di *Lukas Kanade* riesce a tracciare, frame per frame, tali punti senza l'ausili di alcun oggetto esterno, nel caso del *tracking* dell'angolo volante.

Figura 4.2. Condizione di angolo 0°, in rosso si notano i "punti di interesse" selezionati automaticamente dall'algoritmo

Figura 4.3. Condizione di angolo di 40°, l'algoritmo continua a "seguire" i punti nonostante la diversa rotazione

Tale metodo risultava molto comodo e performante per il *tracking* di alcuni punti dell'immagine, ma diventava inusabile nel caso in cui lo spostamento era minimo o, come nel nostro caso, era dovuto a piccole rotazioni. Non riuscendo perciò ad ottenere una stima accettabile è stata presa nuovamente un'altra strada e ripartire con il tracking dei markers sullo sterzo. Il metodo vero e proprio verrà spiegato in dettaglio **??**, in parole povere sfrutta un algoritmo di *Procrustes* [6] che riesce a confrontare due set di posizioni sullo spazio (corrente e precedente) e calcolare l'angolo risultante dalla differenza tra le posizioni dei medesimi punti. Il metodo, associato con un algoritmo di tracking *intelligente* che riesce a tenere un riferimento ai marker del frame precedente, è risultato essere molto performante e particolarmente affidabile in casi *ideali*, durante le prove in auto invece non risulta ancora perfettamente utilizzabile allo stato attuale, poiché si basa molto sullo stato precedente del sistema per l'identificazione dei markers, quindi i problemi principali sono dovuti a

- Variazioni di luminosità all'interno della vettura
- Falsi positivi rilevati a causa della luce riflettente sul volante

- Perdita di visibilità dei markers nel momento in cui la manovra con colpo di sterzo obbliga il conducente a tenere il braccio o la mano davanti ai markers, e quindi ad oscurarne uno o più per qualche secondo
- Cambiamenti improvvisi di rotazione del volante, supporta rotazioni massime di 70-80° al secondo

In tutti questi casi l'algoritmo in qualche modo non riesce a stimare in maniera accettabile l'angolo volante, risultando in un calcolo sbagliato dello stesso o nullo. Il problema si accentua nel tempo, poiché dipendendo dallo stato precedente del sistema l'errore va via via propagandosi senza riuscire a recuperarlo, tranne in certi casi che verranno esposti successivamente.

Capitolo 5

Ambiente di Sviluppo

In questo ultimo capitolo della prima parte introduttiva porremo maggiore attenzione sugli aspetti $informatici$ dello sviluppo di questo lavoro, cominciando ad introdurre la piattaforma Android, le fondamenta della programmazione in ambiente Android, qualche accenno sulla libreria utilizzata per il calcolo dell'angolo volante, e sulla computer vision in generale, terminando con le fasi finali dello sviluppo di un'applicazione vera e propria.

5.1 Android

Android non è solamente un sistema operativo ma include anche una piattaforma software base, come ad esempio browser web, fotocamera; tali applicazioni sono dette *native*, poiché integrate all'interno del sistema operativo. Le applicazioni sviluppate da programmatori esterni al team di sviluppo Android sono invece applicazioni $third-party$; da precisare che non esiste distinzione tra le due, poiché entrambe possono accedere al livello *hardware* del dispositivo tramite API specifiche, entrambe sono sviluppate con lo stesso *Software Development Kit (SDK)*, rilasciato gratuitamente sul web. Android è nato nel 2007, fondato da *Open Handset Alliance (OHA)*, un gruppo di compagnie a cui capo vi è *Google*, tant'è che lo sviluppo principale della piattaforma è proprio in mano a quest'ultimo, il quale annualmente rilascia una nuova versione di Android, corredata da tutti gli aggiornamenti per gli strumenti di sviluppo per supportare quest'ultima. E' basato sul *Kernel Linux*(4.x.x), ed è sviluppato seguendo lo schema illustrato sotto. Essendo un sistema operativo mobile, deve essere in grado di girare su qualunque macchina, quindi anche su un hardware piuttosto limitato: per questo motivo è stato pensato per girare in modo *non* nativo sul sistema ma di appoggiarsi ad una macchina virtuale; dapprima **DVM**, Dalvik Virtual Machine, adesso **ART**, Android RunTime. Altro aspetto fondamentale di Android è il fatto che si tratta di un sistema operativo *aperto*, chiunque voglia può consultare il codice sorgente dell'intero sistema e contribuire allo sviluppo o alla documentazione. Essendo la licenza basata su *Open Source Apache License*, i vari vendor possono customizzare e modificare il sistema operativo a loro piacimento per poi ridistribuirlo sviluppando anche estensioni proprietarie su di esso. Chiunque voglia mettere in commercio un terminale non deve pagare alcuna tassa per utilizzare Android.

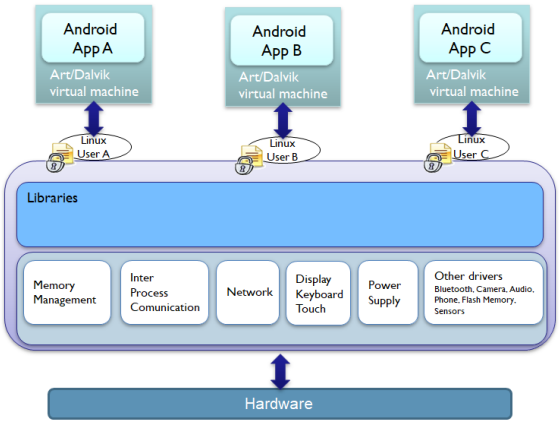


Figura 5.1. Architettura del sistema operativo Android.

Il kernel funziona come uno strato di comunicazione tra il livello fisico del dispositivo e il livello software (in inglese è chiamato *HAL*, Hardware Abstraction Level). Ogni applicazione Android che gira sulla macchina possiede una propria istanza della macchina virtuale ART, girando quindi su un processo a parte.

Ogni applicazione è formata da uno o più *componenti* che possono essere:

- Activity
- Service
- ContentProvider
- BroadcastListener

Ognuno dei quali svolge un ruolo ben preciso nell'interazione con l'utente o con il sistema operativo.

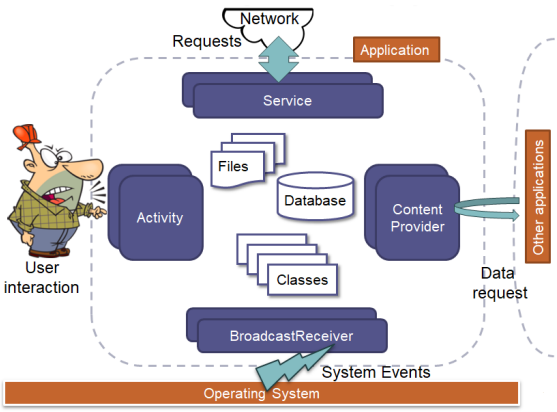


Figura 5.2. Componenti di una applicazione Android.

5.1.1 Android Activities

Una *Activity* rappresenta di solito una schermata interna di una applicazione, viene utilizzata per svolgere un task o per mostrare qualcosa all'utente; può quindi essere interattiva o meno. La cosa importante è che le Activities possiedono una interfaccia grafica con cui l'utente può interagire. Per sviluppare una activity si estende la classe pre-esistente *android.app.Activity*. Il sistema operativo si occupa di gestire il *lifecycle* di ogni activity invocando dei metodi che, durante lo sviluppo di una activity, vanno *sovrascritti*:

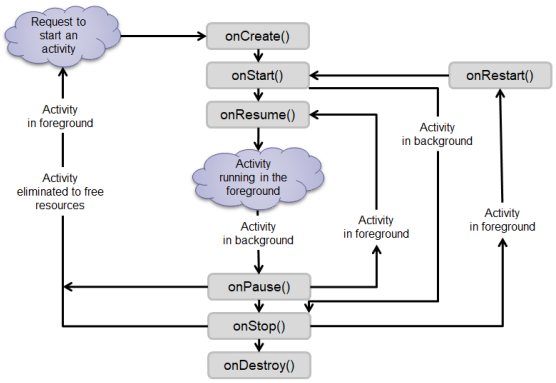


Figura 5.3. Lifecycle di una activity.

- `protected void onCreate(Bundle b)`,

  il metodo è invocato quando l'activity è create per la prima volta o quando è rilanciata dopo essere stata interrotta (il parametro *Bundle* rappresenta un costrutto utile per ripristinare lo stato precedente). All'atto dell'invocazione di questo metodo di solito viene svolta la fase di inizializzazione dei dati, acquisizione di risorse permanenti e definizione di *callback*.

```
public class ExampleActivity extends Activity {
  @Override
  public void onCreate(Bundle b) {

    super.onCreate(b);

    //Create and initialize a view
    TextView tv=new TextView(this);
    tv.setText("Hello Android");

    //show the view content
    setContentView(tv);
  }
}
```

- `@Override`
  `protected void onStart()`,

è invocato quando diventa visibile l'interfaccia utente. Di solito non viene ridefinito questo metodo.

- `@Override`
  `protected void onResume()`,

  è invocato quando l'activity si trova in alto nello *stack* delle activities, e quindi diventa interattiva per l'utente. Di solito viene utilizzato per avviare animazioni o video, oppure per acquisire risorse temporanee.

- `@Override`
  `protected void onPause()`,

  è invocato quando l'activity deve essere spostata in seconda posizione nello stack delle activities, di solito vengono rilasciate le risorse temporanee ed interrotti video e animazioni. Se devo lanciare un'altra activity questa non verrà avviata finché non termina il metodo.

- `@Override`
  `protected void onDestroy()`,

  è invocato quando l'activity viene terminata e rimossa dalla memoria. Può essere invocato dall'utente tramite il metodo $finish()$.

5.1.2 Android Services

Un *Service* rappresenta invece un un costrutto ideato per girare in background, senza che l'utente se ne accorga; non possiede un'interfaccia grafica e quando viene lanciato esegue di solito un task abbastanza corposo e ripetitivo, può ad esempio riprodurre musica in background o fare il $fetch$ di dati nel network. Esistono due diversi tipi di Service:

- **Started Service**, quando è una activity ad invocare un service, di solito gira finché non termina il task e dopo muore senza ritornare alcun valore.
- **Bound Service**, quando una activity si associa ad un servizio; permette una comunicazione $client - server$ tra i due e nel momento in cui l'activity viene distrutta anche il service viene distrutto. Il ciclo di vita del service in questo caso è legato ai componenti che sono associati ad esso.

I metodi fondamentali della classe *android.App.Service*:

- `@Override`
  `void onStartCommand(...)`,

  è il metodo invocante di un service di tipo *started* e una volta chiamato avviene la fase di inizializzazione e di avvio di un service. Questo rimane in esecuzione fin quando non viene arrestato di proposito.

- `@Override`
  `void onBind(...)`,

è il metodo chiamato dal sistema quando un componente si lega ad un service di tipo *bound*, invocando la *bindService*(...).

- `protected void onCreate(...),`
  `protected void onDestroy(...),`

  sono invocati dal sistema quando il service viene inizializzato, poco prima della chiamate *invocante*, o quando il service cessa la sua attività per rilasciare le risorse in maniera pulita.

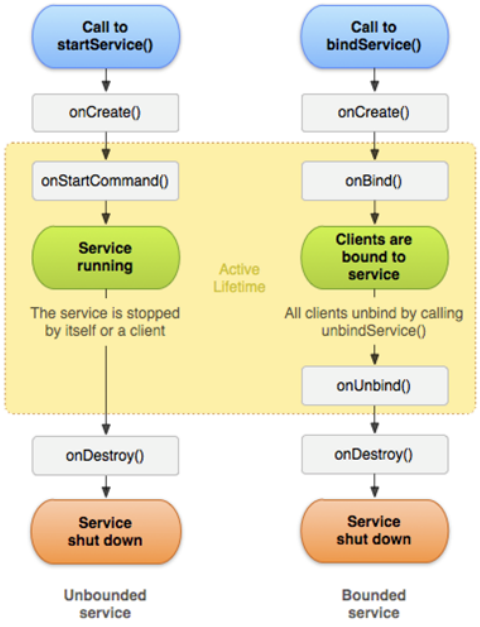


Figura 5.4. Lifecycle dei due tipi di Service.

5.1.3 Android Broadcast Receiver

Un *Broadcast Receiver* permette invece ad una activity di mettersi in ascolto in attesa di eventi di sistema, come vedremo verrà usato per leggere i valori dei sensori del terminale ad esempio. Viene definito estendendo la classe *Android.App.BroadcastReceiver*, ed implementando il metodo:

```
@Override
public void onReceive(Context context, Intent intent),
```

tale metodo permette di definire il comportamento dell'applicazione nel momento in cui riceve il dato per cui si era precedentemente posta in ascolto. Il metodo ha come argomento un oggetto *Intent*, che contiene le i dati richiesti ed inviati dal mittente (nel caso dei sensori i valori grezzi).

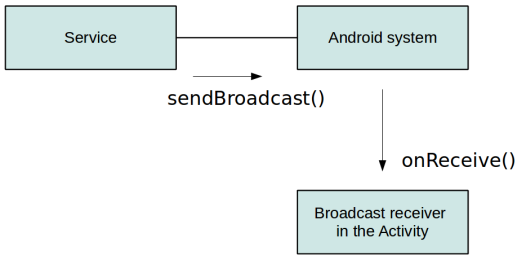


Figura 5.5. Funzionamento di un BroadCast Receiver.

5.1.4 Android Persistent Data

In ambito Android esistono sostanzialmente due metodi per poter memorizzare dati: *server remoto*, *dati locali*. Durante lo sviluppo dell'applicativo non è stato necessario interfacciarsi con qualsiasi server per poter richiedere/salvare dati, quindi andremo a trattare solamente i metodi di memorizzazione in locale. Esistono principalmente tre metodi di memorizzazione in locale(andremo ad analizzare solamente i due utilizzati):

- **Preferences:** rappresenta il meccanismo base di salvataggio informazioni; utilizza un costrutto di $key - value$(mappa) interno, e il file corrispondente può essere utilizzato anche da altre applicazioni. Il tutto viene scritto su un file di tipo *.xml*, quindi facilmente leggibile anche da altri applicativi. Può anche essere creato un file accessibile solamente dall'applicazione che lo crea specificando il parametro. Il costrutto è il seguente:

  ```
  SharedPreferences settings = getSharedPreferences("MyPreferences",
                                                    MODE_PRIVATE);
  ```

  Per poter inserire un valore, e quindi modificare l'oggetto bisogna innanzitutto creare una reference all'oggetto statico *Editor*, dopo invocare il metodo apposito *edit*() seguito da un *apply*() per confermare le modifiche. Di seguito un frammento che mostra un esempio di utilizzo:

```
public static final String PREFS = "MyPreferences";
  private String password;

  @Override
  protected void onCreate(Bundle b) {
    super.onCreate(b);
    //...
    SharedPrefereces sp= getSharedPreferences(PREFS, MODE_PRIVATE);
    password = sp.getString("password", "");
  }

  @Override
  protected void onStop() {
    super.onStop();
    SharedPrefereces sp= getSharedPreferences(PREFS, MODE_PRIVATE);
    SharedPreferences.Editor e=sp.edit();
    e.putString("password", password);
    e.commit();
  }
```

- **Database Storage** il metodo "classico" di memorizzazione dei dati, implementato in Android tramite l'utilizzo di *SQLite*. Per poter utilizzare un database costruito all'esterno dell'applicazione bisogna importare il file all'interno della $assets - folder$ e poi andare a leggerlo tramite i comandi classici:

  ```
  SQLiteDatabase mDatabase;
  mDatabase = openOrCreateDatabase("my_sqlite_database.db",
                           SQLiteDatabase.CREATE_IF_NECESSARY, null);
  ```

  Per le *queries* è possibile utilizzare i metodi forniti dalla classe o eseguire dei comandi *RAW*, direttamente come mostrato di seguito:

  ```
  String cmd="CREATE TABLE tbl_authors ( "+
                       "id INTEGER PRIMARY KEY AUTOINCREMENT, "+
                       "firstname TEXT, "+
                       "lastname TEXT);"
  mDatabase.execSQL(cmd);
  ```

  O in modo "automatico":

  ```
  ContentValues values = new ContentValues();
  values.put("firstname", "J.K.");
  values.put("lastname", "Rowling");
  long newAuthorID = mDatabase.insert("tbl_authors", null,
                                      values);
  ```

5.1.5 Android Camera

Allo stato attuale in Android vi sono due famiglie di API disponibili per accedere all'hardware fotocamera del dispositivo:

- **android.hardware.Camera**, la **vecchia** classe di metodi utilizzata per accedere alle funzioni principali della fotocamera, non supporta molte funzioni ormai presenti in molti dispositivi di fascia medio-alta del mercato (ZSL, HDR shot, per-frame contol, RAW data access). Supporta 3 modi principali di operazione:
  - Preview
  - Image Capture
  - Video Recording

  E' possibile effettuare la configurazione della fotocamera tramite la classe *CameraParameters*. L'intera famiglia di API è ormai deprecata da qualche anno a causa di queste limitazioni.
- **android.hardware.camera2**, la nuova classe di API disponibile in Android; rivoluziona il modo di elaborare l'immagine tramite un processo a *pipeline*: riceve in input delle richieste per ogni singolo frame da elaborare, esegue l'operazione di *capture* per ognuna di queste richieste ed emette, in output, un file di *metadata* contenente informazioni sul frame e un set di immagini bufferizzate. Il processo avviene seguendo l'ordine di ingresso delle richieste di esecuzione che arrivano in input. Tale meccanismo permette di elaborare ogni singolo frame richiesto, ed applicare i controlli e le personalizzazioni volute.

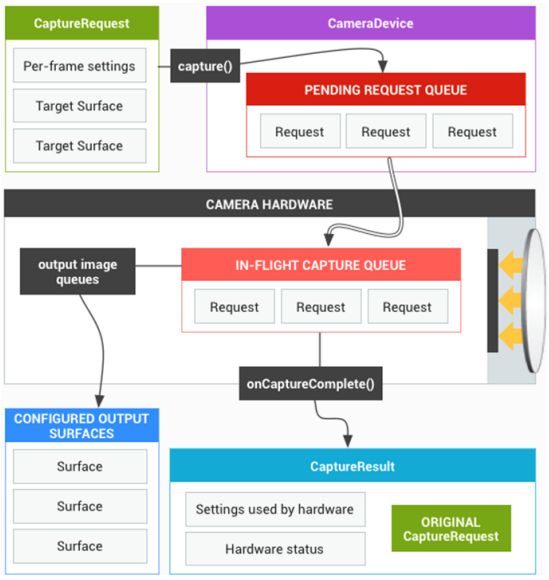


Figura 5.6. Rappresentazione della pipeline.

Durante il lavoro svolto è stata utilizzata la prima famiglia di API, anche se deprecata ufficialmente, per necessità legate all'implementazione di *OpenCv*, basata appunto sulla prima versione di API della camera. Ciò ha posto dei limiti su alcuni aspetti dell'algoritmo sviluppato per il calcolo dell'angolo volante, come ad esempio non poter in alcun modo

modificare il valore del *framerate* della registrazione video durante le prove, o la possibilità di poter switchare facilmente l'accesso alla camera tra le diverse classi che la utilizzano.

5.1.6 NDK e JNI

Con NDK (Native Development Kit) si vuole indicare un set di strumenti, rilasciati tramite un intero pacchetto all'interno dell'IDE *Android Studio*, che permettono di utilizzare parti di codice scritte in linguaggio nativo (C o C++) direttamente all'interno dell'applicazione Android. Questo tool permette quindi di poter implementare parti di codice, librerie o intere classi precedentemente scritte in un altro linguaggio di programmazione, direttamente all'interno del codice Java; ciò ovviamente non comporta una diminuzione delle performance sui terminale anzi, essendo codice non interpretato (teoricamente, poiché in realtà il codice nativo passa poi per delle interfacce per comunicare con la controparte in Java) permette di mantenere le stesse performance. Android permette di utilizzare tale codice tramite JNI (Java Native Interface), uno strumento che permette di di realizzare una *corrispondenza* tra codice nativo e chiamate a funzioni Java. Permette infatti di poter chiamare codice nativo all'interno di metodi Java o, al contrario, essere chiamato da funzioni native. La complessità sta nel fatto che, essendo i tipi differenti, esiste una *Mappa* Java <-> Codice nativo per i tipi fondamentali, mentre per i tipi definiti in fase di realizzazione del programma il tutto viene tramite l'incapsulamento all'interno di una struttura *conosciuta* dal codice nativo, **JNIEnv**, tale strutta viene utilizzata come puntatore all'interno dei metodi ed inizializzata tramite i tipi Java, chiamando le apposite funzioni di conversione.

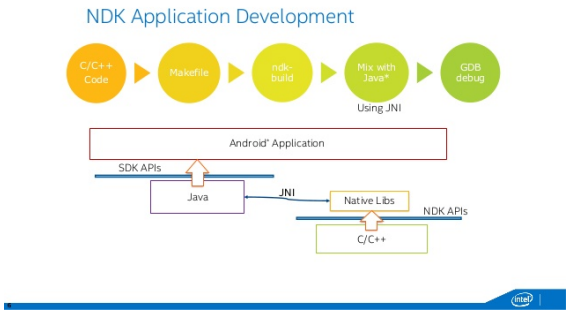


Figura 5.7. Processo di funzionamento NDK.

All'interno dell'applicazione non verranno implementate funzioni native ma la spiegazione relativa è utile poiché la libreria *OpenCv* è implementata in Android tramite NDK.

5.2 Libreria OpenCv

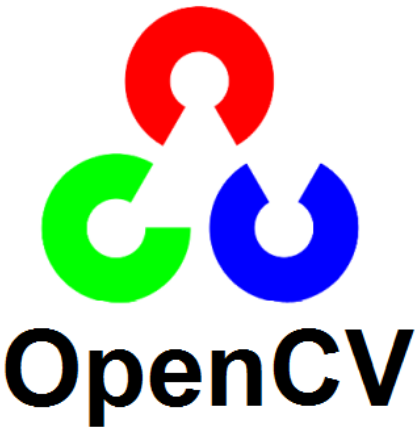


Figura 5.8. Logo di OpenCv.

[8] OpenCv è una libreria open source sviluppata in C++, pensata per essere supportata da tutti i dispositivi dotati di un hardware consono ad eseguire calcoli matematici a volte molto complessi, ma nello stesso tempo per essere il più possibile efficiente dal punto di vista delle performance. E' stata sviluppata per essere utilizzata in applicazioni realtime, motivo per cui è stata scritta in linguaggio C, ottimizzato e veloce, sopratutto sfruttando l'esecuzione parallela dei *task*. OpenCv mira a fornire al programmatore un set di metodi molto efficaci dal punto di vista della complessità che permettono, tramite poche righe di codice, di eseguire dei task molto complessi; contiene infatti moltissime funzioni applicabili in contesti differenti: dal *tracking* al riconoscimento automatico, contiene inoltre una libreria che supporta l'apprendimento automatizzato (*Machine Learning*).

5.2.1 Installazione e Configurazione

//PRENDERE LA PARTE DI ROBERTO

Per quanto riguarda Android, OpenCv utilizza un particolare costrutto molto utile al programmatore, che ricopre il ruolo di un vero e proprio servizio interno che gestisce in tutto la parte di *implementazione* di OpenCv: **OpenCV Manager** utilizza direttamente i binari disponibili nei file della libreria per rendere il tutto molto più performante e sicuro, poiché in caso di *update* basta cambiare solamente una stringa all'interno dell'applicazione per sostituire la versione corrente utilizzata. In poche righe di codice è possibile inizializzare l'intera piattaforma OpenCv e controllare che il tutto sia installato e funzionante tramite delle chiamate asincrone che vanno effettuate durante la fase iniziale del lancio dell'activity.

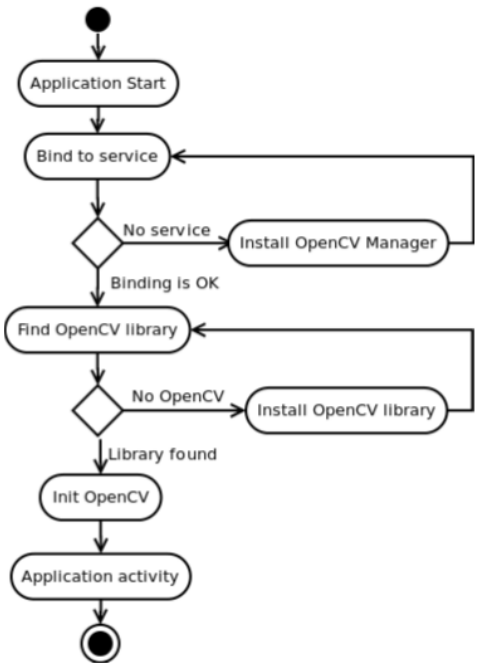


Figura 5.9. Flow-Chart che rappresenta il flusso di esecuzione di OpenCv Manager.

5.2.2 Il sistema di riferimento

A differenza della maggior parte dei sistemi di grafica, OpenCv utilizza un sistema di riferimento con asse Y invertito, come mostrato in figura.5.2.2

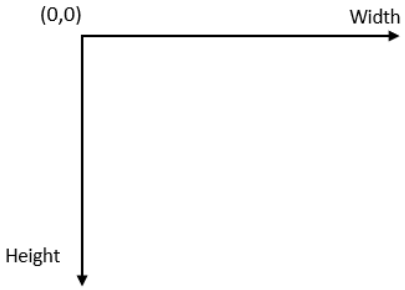


Figura 5.10. Sistema di riferimento OpenCv.

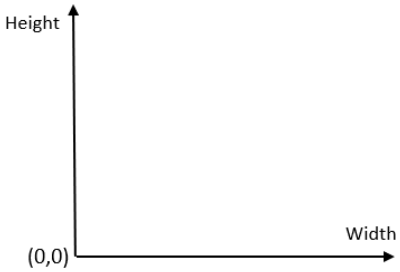


Figura 5.11. Sistema di riferimento in ambiente Android.

Ciò comporta che tutti i calcoli effettuati su altri editor o rispetto ad un sistema di riferimento tradizionale, come quello utilizzato da Android, ha comportato una maggiore attenzione durante la stesura del codice per modificare appunti l'orientamento rispetto al sistema di riferimento OpenCv. Inoltre c'è da considerare che il porting della libreria su

android permette di utilizzare la *preview* del flusso di immagini solamente in modalità *landscape*, ciò vuol dire che tutta l'interfaccia che viene implementata nell'activity della preview deve essere sviluppata seguendo la logica *orizzontale*.

5.2.3 Modelli di Colore

[8] Come sappiamo il cervello umano riesce a processare un'immagine generica poiché nel corso degli anni *memorizza* informazioni circa particolari forme o colori e tramite queste riesce a riconoscere o ad associare oggetti ai *tipi* fondamentali che memorizza. Tutto ciò invece non succede per le macchine in generale, le quali invece associano ad ogni immagine memorizzata o processata tramite il sensore ottico, una semplice *matrice di valori*; tali valori non sono composti solamente da un numero bensì da:

- **posizione**
- **colore**
- **opacità**

Queste informazioni sono contenute all'interno di un singolo elemento chiamato *Pixel*. Una generica immagine viene quindi rappresentata mediante una matrice di pixel, i valori associati ad ogni elemento interno di un pixel invece possono variare a seconda della *profondità* dell'immagine. Lo standard utilizzato per la rappresentazione dei colori è tutt'ora la codifica **RGB**(*Red, Green, Blue*), di solito utilizzato con una intensità per ogni valore che va da 0 a 255, ogni colore viene quindi rappresentato da un numero binario a 32 bit, diviso in 4 parti: le prime tre parti rappresentano i rispettivi colori rosso, verde, blu, l'ultimo quartetto invece viene utilizzato per il valore *alpha*, opacità del colore.

Figura 5.12. Immagine rappresentata in RGB - Scala di Grigi - Binario.[11]

Un altro standard di largo utilizzo è quello **HSV** o *HSB*, *Hue, Saturation, Value*; in cui il primo valore rappresenta il concetto proprio di *tonalità*, il secondo di *tinta* e il terzo di *intensità*, è quindi possibile rappresentare diversi colori modificando solamente uno dei tre parametri di rappresentazione. Esiste una terza rappresentazione *HSL* che utilizza invece la *luminosità* come terzo parametro di rappresentazione.

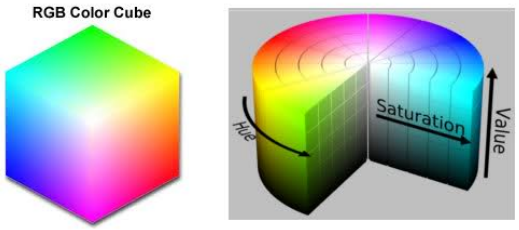


Figura 5.13. Rappresentazione del colore tramite RGB & HSV.

5.2.4 Funzioni base utilizzate nell'algoritmo di calcolo dell'angolo volante

OpenCv mette a disposizione numerose funzioni di manipolazione dell'immagine: ciò permette di poter applicare delle trasformazioni a ogni frame processato dal sensore della fotocamera. L'obiettivo è quello di poter ricavare, frame dopo frame, delle caratteristiche dell'immagine che si mantengono costanti; nel nostro caso per riuscire ad invidiare i *markers* si deve in qualche modo eliminare quella parte *inutile* del frame, ovviamente non in modo fisico ma applicando dei filtri per mantenere solo le informazioni necessarie. Nel nostro caso queste corrispondono a dei cerchi bianchi su sfondo nero, quindi il risultato finale del processo di manipolazione vede l'isolamento dei *bianchi* sui neri e la massima riduzione del disturbo dell'immagine. Di seguito sono elencate le principali trasformazioni utilizzate, applicate seguendo questo ordine:

- **Color Transformation**: è applicato tramite la funzione integrata in OpenCv, **cvtColor(..)**; permette di cambiare lo spazio di colore di una immagine in un altro, nel nostro caso in scala di grigi. Il tutto viene fatto specificando l'immagine di input, quella di output, la conversione da effettuare tramite un codice ($CV\_RGBA2GRAY$) secondo la relazione:
$$Y < -0.299R + 0.587G + 0.114B \tag{5.1}$$

- **Gaussian Blur**: il filtro di sfocatura è utilizzato nel nostro caso per ridurre il rumore dell'immagine che potrebbe in qualche modo *ingannare* il nostro algoritmo, ad esempio con piccole macchie bianche all'interno. Per eseguire la sfocatura viene utilizzato un *Filtro Gaussiano*: il filtro è realizzato applicando la convoluzione a ciascun punto in input con un $Kernel$ Gaussiano, alla fine vengono sommati tutti i valori per estrapolare il risultato. La curva Gaussiana è rappresentata come:
$$G_0(x, y) = Ae^{\frac{-(x-\mu x)^2}{2\sigma_x^2} + \frac{-(y-\mu_y)^2}{2\sigma_y^2}} \tag{5.2}$$

- **Threshold filter**: il filtro viene utilizzato letteralmente per filtrare i pixel che hanno un valore maggiore di una certa *soglia*; nel nostro caso per considerare tutto ciò che è chiaro come bianco, il resto nero. Il filtro può essere applicato in maniera lineare o inversa, specificando l'argomento opportuno nella funzione.

Figura 5.14. Applicazione in sequenza delle funzioni elencate sopra per riuscire ad evidenziare ed isolare le aree di bianco. La prima immagine rappresenta lo stato iniziale. [13]

Durante lo sviluppo dell'algoritmo di tracking per il calcolo dell'angolo volante sono state utilizzate queste funzioni per seguire la logica dell'*Object Recognition:* riuscire a far capire, frame dopo frame, che nell'immagine sono ancora presenti dei punti d'interesse (*markers*), nonostante l'immagine risulti leggermente spostata, lo sterzo risulti inclinato o le condizioni di luce siano leggermente variate. Successivamente [**?**], sono stati rappresentati i principali step di avanzamento dell'algoritmo.

5.2.5 Algoritmo di Tracking per l'angolo volante

In figura è stato rappresentato, tramite diagramma a blocchi, lo schema esecutivo dell'algoritmo, anche se in modo molto semplificato. Per andare un po' più nel dettaglio andremo a descrivere la fase di esecuzione che si occupa di dare un feedback all'utilizzatore sullo stato di esecuzione dell'algoritmo (quella che comincia a sinistra, dal **Main Thread**, e la fase in *background*, a destra, che effettivamente non viene per nulla percepita dall'utente ma che in realtà svolge tutti i calcoli per riuscire a calcolare l'angolo volante. Le due fase non lavorano in maniera sincronizzata anzi, il costrutto utilizzato è quello del pattern **"Produttore - Consumatore"**; un thread inserisce i frame da processare in una *coda* thread-safe e l'altro si occupa invece di prelevare i frame, analizzarli e poi scartarli. Tale meccanismo permette di non intaccare l'esperienza utente sull'activity con rallentamenti o attese poiché il tutto è realizzato in maniera *asincrona*.

C'è da dire che questo schema rappresenta il flusso in esecuzione durante la fase di *preparazione*, ossia una volta impostato il tutto e verificato il corretto funzionamento vengono salvati i parametri di luminosità e contrasto iniziali, la posizione della *ROI* [1] , la posizione iniziale dei markers. Questi vengono poi passati al *Service* che utilizza la fotocamera in background per processare l'angolo volante, che in sostanza replica quello fatto

[1] letteralmente *Region of Interest*, rappresenta la porzione di immagine che viene effettivamente considerata dall'algoritmo, nel nostro caso è possibile muovere e/o ridimensionare la *ROI* tramite le gestures di default.

in questo schema ma sotto forma di servizio Android, proprio per riuscire a girare in background. La differenza sostanziale sta nel fatto che non si è potuto utilizzare *direttamente* OpenCv, poiché si appoggia su un oggetto ($JavaCameraView$) che deve essere necessariamente implementato in una *Activity* (estende un oggetto di tipo $PreviewCallback$). Per questo motivo il service responsabile del tracciamento in background utilizza un oggetto di tipo $Hardware\ Camera$, che va ad implementare la stessa interfaccia di cui si serve OpenCv, ma il tutto è stato fatto manualmente interfacciandosi con la fotocamera.

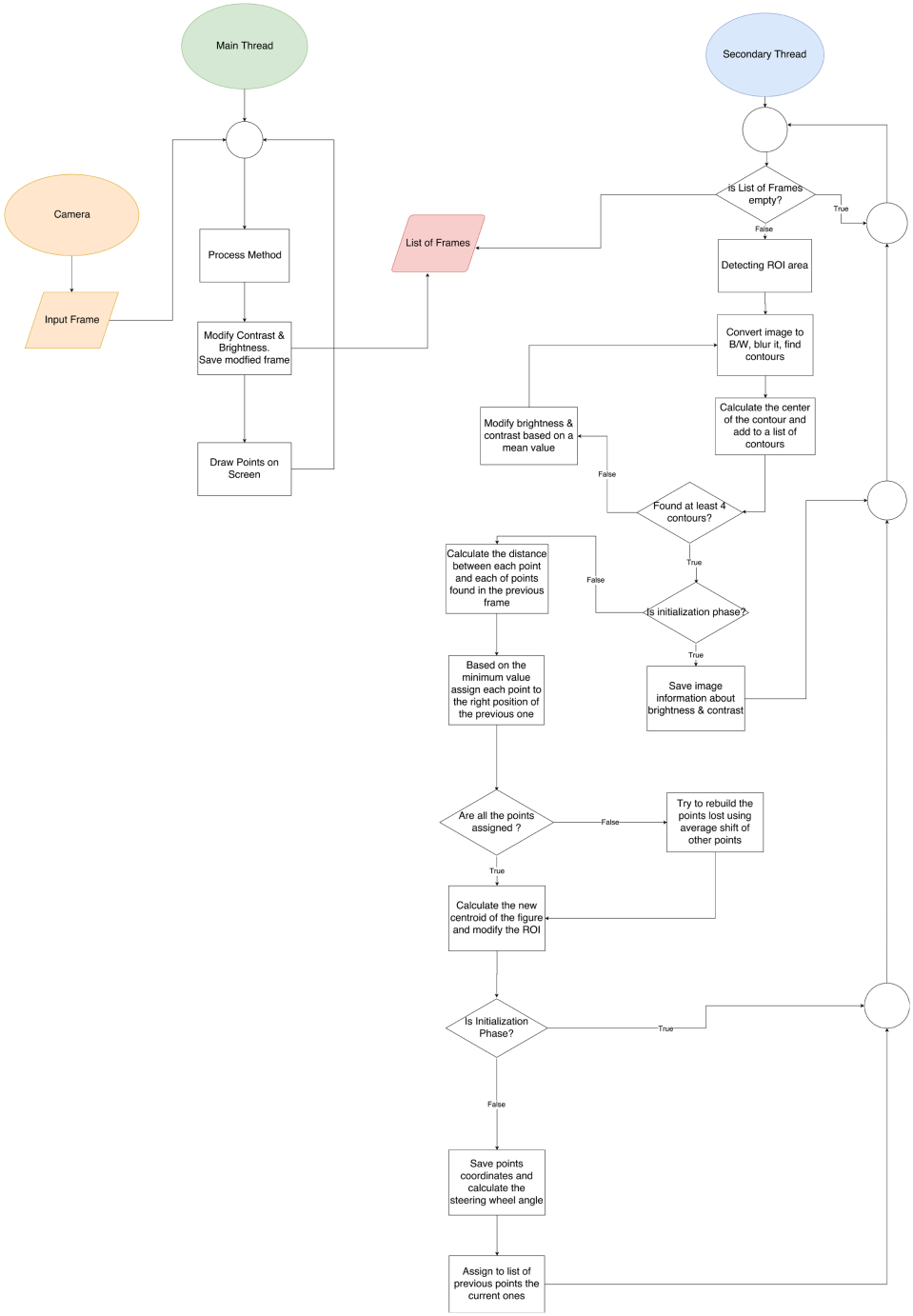


Figura 5.15. Workflow semplificato dell'algoritmo di calcolo dell'angolo volante. A sinistra è mostrata l'esecuzione in foreground, a destra quella in background.

Fase in Foreground:

In questa fase avviene il salvataggio dei frame nella coda asincrona; il metodo principale, che è una *callback* implementata nell'interfaccia della classe OpenCv, riceve il frame come un oggetto Matrice, lo salva in una variabile e su questa effettua degli aggiustamenti di luminosità e contrasto: i valori vengono prelevati dalla classe *statica* che invece implementa la parte in background dell'algoritmo, come vedremo dopo. Dopo il salvataggio del frame vengono stampati sullo schermo i *markers* riconosciuti e il valore dell'angolo volante, se l'algoritmo è già in esecuzione. Quando il metodo ritorna viene immediatamente richiamato poiché la callback viene invocata ogni qualvolta vi è un nuovo frame da processare (circa 25 volte al secondo).

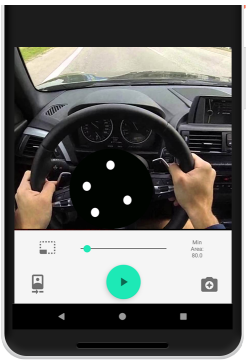


Figura 5.16. Activity di acquisizione dell'angolo volante

Fase in Background:

La fase in background è gestita da un thread secondario che ha il compito di prelevare inizialmente il frame dalla lista, se questo non è presente grazia al costrutto utilizzato per la lista, *LinkedBlockingQueue*, un costrutto bloccante [2] che consente di mantenere un certo livello di performance anche se la coda risulta vuota. Una volta prelevato il frame questo viene elaborato, seguendo la logica illustrata nello schema, a seconda che lo stato sia di *Inizializzazione* o meno.

• Nel primo caso l'algoritmo non fa altro che restituire tutte le coordinate di *aree* di bianco trovate e il thread principale, in esecuzione parallela, si occupa di disegnarle sullo schermo per dare un feedback all'utente. Inoltre nel momento in cui non vengono individuato almeno quattro punti, questo prova a modificare i parametri di luminosità e contrasto basandosi su informazioni memorizzate in casi ottimali: l'algoritmo nel momento in cui individua quattro punti calcola, tramite un algoritmo specifico che lavora sul calcolo del valor medio di luminosità dell'immagine, il valore di luminosità e lo aggiunge ad un array

[2]Vuol significare che il thread rimane in attesa del dato nel momento in cui la coda è vuota, senza consumare cicli inutili di CPU che farebbero andare in sovraccarico il programma nel momento in cui la coda restasse vuota per molto tempo. Tale costrutto è implementato direttamente all'interno della struttura coda, ed è sviluppato utilizzando il pattern di *Signal-Wait* a basso livello.

di valori e su questo viene calcolata la *media mobile*: in base al valore medio nel momento in cui mancano dei punti, l'algoritmo modifica il valore fin quando questo non raggiunge il valore medio; in questo modo si compensa al fatto che per qualche frame l'immagine può risultato più scurita o più chiara, e nonostante ciò l'algoritmo si *arrangia* ad aggiustare l'immagine in questo modo. Per quanto riguarda la ricerca delle aree di bianco questa viene fatta utilizzando una funzione integrata in openCv, *findCountours(...)*, restituisce una lista di oggetti matrice in cui ogni ad ogni matrice corrisponde un *contorno* trovato: iterando sulla lista si riesce a filtrare i contorni interessati tramite un filtro basato sul calcolo dell'area del contorno. L'area viene calcolata utilizzando la funzione *Moments* di OpenCv [15]. L'area minima da soddisfare viene invece scelta dall'utente tramite l'apposita *SeekBar*, come si evince dallo screen sotto [**?**]. Tutto questo viene fatto per ogni frame fin quando l'utente, individuati i quattro punti di interesse, non preme il pulsante *play* sul display; a quel punto vengono salvate le coordinate dei quattro punti in una lista e si passa al secondo caso, quello di *Recording*.

Figura 5.17. In basso vengono mostrati i controlli disponibili per modificare i parametri di luminosità e contrasto.

• La differenza sostanziale è che qui avviene il calcolo dell'angolo volante, come accennato poc'anzi, tramite l'algoritmo di **Orthogonal Procrustes**: innanzitutto bisogna dire che ad ogni frame avviene un confronto fra i punti appena calcolati e i punti al frame precedente, se questi erano quattro, o comunque rispetto ai punti individuati nell'ultimo frame *utile*. Il confronto viene effettuato calcolando la distanza tra ogni punto appena individuato e ogni punto appartenente al frame utile precedente; in questo modo si riesce ad assegnare ogni punto, in ordine, al corrispettivo precedente e ciò è fondamentale poiché per il calcolo dell'angolo tale relazione deve essere rispettata. Nel momento in cui questa assegnazione non viene fatta, quindi letteralmente *si perde* un punto, questo viene *ricostruito* tramite una procedura di approssimazione della posizione del punto: si assegna, al punto perso, il valore di spostamento uguale allo spostamento medio degli altri punti e di direzione pari all'ultima direzione individuata precedentemente; nonostante sia un'approssimazione piuttosto grezza, essendo l'intervallo di campionamento molto alto, si riesce a ricostruire in modo accettabile se il movimento non viene *perso* per molti frame consecutivi. Una volta assegnati i punti si procede come nel caso precedente, salvando le informazioni circa la luminosità dell'immagine e si applica l'algoritmo di calcolo dell'angolo

volante. Come funziona?

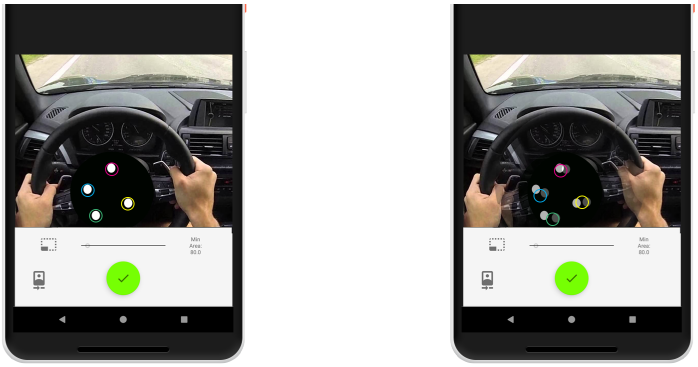


Figura 5.18. Nella seconda immagine i punti, in fase di rotazione, risultano spostati rispetto alla posizione iniziale: l'algoritmo riesce a *seguirli*.

• L'algoritmo di **Procrustes**: lo sviluppo dell'algoritmo è basato sul problema di *Orthogonal Procrustes Problem*, in cui si vuole trovare una matrice ortogonale R che approssima al meglio la *mappatura* di una matrice A con una matrice B. Il problema è risolvibile considerando una ulteriore matrice:

$$M = BA^T \tag{5.3}$$

Per trovare tale matrice R è necessario applicare la SVD (**Singular Value Decomposition [16]**: permette di fattorizzare una matrice M nella forma:

$$M = U\Sigma V^T \tag{5.4}$$

con U matrice $m \times m$ *unitaria*, Σ matrice $m \times n$ *rettangolare diagonale* e V matrice $n \times n$ *unitaria*. I valori diagonali di Σ sono i cosiddetti **valori singolari** di M.

R può allora essere scritta come

$$R = UV^T \tag{5.5}$$

Tale ragionamento viene applicato per calcolare la rotazione di due set di punti, un set comprende i punti del frame corrente, l'altro quelli del frame precedente, nonostante questi non siano perfettamente allineati allo stesso sistema di riferimento.

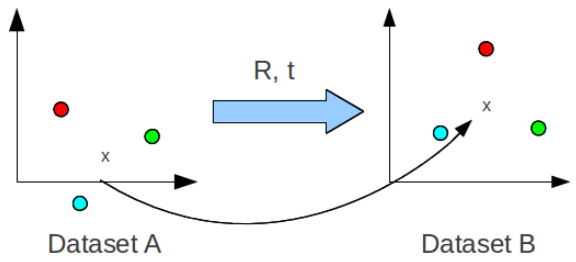


Figura 5.19. Rappresentazione dei due set di punti e della trasformazione.

Nella prima parte viene quindi calcolato il *centroide* di entrambi i set di punti e vengono quindi riportati all'origine, così di evitare problemi con la traslazione, e considerare invece solamente una trasformazione, la *rotazione*.

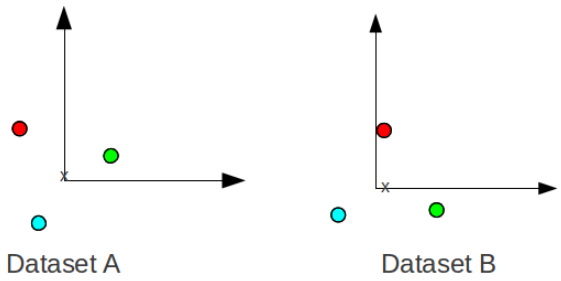


Figura 5.20. Normalizzazione dei due set di punti rispetto all'origine.

Per calcolare quest'ultima viene estrapolato l'angolo di rotazione θ dalla matrice R come:

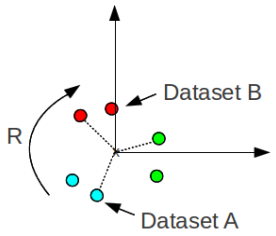


Figura 5.21. Rappresentazione di R rispetto ai due set di punti.

$$\boxed{\theta = atan(\frac{R_{10}}{R_{00}})(\frac{180}{\pi})} \tag{5.6}$$

Essendo R una matrice di rotazione rappresentata come:

$$R = \begin{bmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

L'angolo restituito rappresenta il δ dell'angolo frame dopo frame, quindi l'angolo volante totale viene inteso come

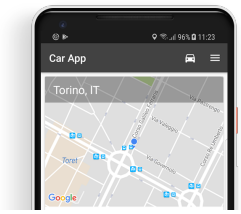
$$\boxed{\Theta = \sum_{1}^{n} \delta\theta} \tag{5.7}$$

Parte II

Seconda Parte

Capitolo 6

Applicazione Android: Car-App



Nelle prossime pagine verrà illustrato il funzionamento dettagliato dell'applicazione sviluppata durante questi sei mesi e commentato, in dettaglio, il codice relativo ad ogni parte dell'applicazione. Ove possibile verranno mostrati dei *flowchart* per spiegare in maniera semplice dei passaggi complessi; inoltre verranno illustrati in dettaglio anche le componenti grafiche utilizzate, le librerie e i layout per tutte le *Activities*.

6.1 File di configurazione

In questa prima e piccola sezione andremo ad illustrare le operazioni e il contenuto generale di tali file, detti di *sistema*, proprio perché sono necessari per interfacciare correttamente l'applicazione al sistema operativo Android. Il loro reale contributo ovviamente non è visibile in fase di esecuzione, ma sono obbligatori per svolgere certe operazioni sopratutto relative alla fase di *build* del progetto e a quella di primo interfacciamento con il sistema operativo.

6.1.1 Il File Manifest

Il file di manifest presente in Android, **AndroidManifest.xml** è un file che viene letto dal sistema operativo prima di *runnare* il codice relativo all'applicazione. Serve sostanzialmente a

- dichiarare il *Package Name* relativo all'applicazione;
- descrivere e dichiarare tutti i componenti dell'applicazione, quali activities, services, providers etc...;
- gestisce quali permessi può richiedere l'applicazione riguardo all'hardware o al software: ad esempio acquisire la fotocamera, leggere i messaggi o il registro chiamate, poter leggere o scrivere file, acquisire la geo localizzazione.

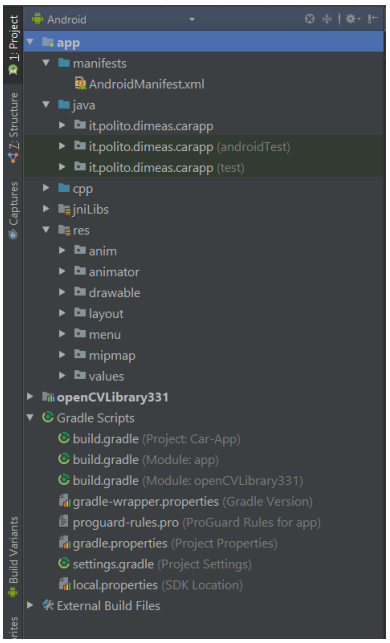
6.1.2 Gradle Build Toolkit

Il sistema di *build* di Android permette di poter compilare separatamente il codice sorgente, le risorse esterne e infine poterle *impacchettare* in un unico file .apk. Tale processo è implementato in Android Studio tramite un tool particolare, **Gradle:** permette di poter scrivere un file di configurazione in cui è possibile modificare il processo standard e quindi includere librerie esterne, modificare i parametri di build ed eventualmente riutilizzarlo in altre applicazioni. I file associati al processo di build all'interno dell'applicazione sono tre:

- build.gradle (*Top Level*): tale file permette di specificare una configurazione di build da applicare a tutti i moduli all'interno dell'applicazione. Di solito non viene modificato alcun parametro all'interno
- build.gradle(*Module Level*): è un file legato ad ogni *modulo* dell'applicazione e permette di poter configurare il tipo di build da effettuare e le eventuali librerie esterne.
- gradle.properties: è un file creato appositamente dall'ambiente di sviluppo in cui sono memorizzati alcuni parametri relativi all'ambiete di build. Nella maggior parte dei casi non vengono mai modificati.

6.2 La struttura dell'Applicazione

Di seguito è mostrata la struttura interna dell'applicazione gestita da Android Studio. Come si può facilmente intuire la struttura rappresenta un vero e proprio *Albero*, in cui vi è una *root* da cui tutto parte e da lì si diramano poi le varie componenti.



• **app Folder:** rappresenta la cartella *root* del progetto: sono esclusi solo i file relativi a librerie esterne e moduli aggiuntivi.

• **manifests Folder:** contiene il file $manifest$ descritto poco prima (6.1.1).

• **java Folder:** al suo interno troviamo tutte le classi java del progetto, inclusi i test da effettuare sulle classi. E' possibile creare delle sottocartelle per comodità, ad esempio per raggruppare classi dello stesso tipo.

• **cpp Folder:** questa è una cartella che viene creata dall'utente (nel nostro caso) per poter inserire classi in linguaggio nativo (5.1.6). Nel nostro caso contiene solo i file relativi ad *OpenCv*.

• **jniLibs Folder:** anche questa è appositamente creata dall'utente per poter incorporare librerie in linguaggio nativo *esterne*(nel nostro caso OpenCv). All'interno sono presenti i file binari .a per ogni architettura supportata.

• **res Folder:** questa è la libreria delle *resources*, al suo interno contiene tutte le risorse che andremo ad utilizzare nel codice: animazioni, i file di layout in .xml, i file di menu in .xml, i valori delle stringhe definite e tutte le immagini (*drawable*) utilizzate, incluse le icone.

• **openCv Library:** rappresenta il modulo esterno importato nel progetto Android direttamente dall'IDE, si viene a creare una cartella a parte contenente pressoché la stessa struttura di un'applicazione vera e propria, contiene infatti un file $manifest$ e una cartella *java*, contenente le chiamate alle classi native della libreria come spiegato (5.2.1).

6.3 I principali Algoritmi sviluppati

Gli algoritmi che si andranno a trattare in questa sezione riguardano le approssimazioni principali che vengono fatte durante una qualsiasi prova, sono stati sviluppati da *zero* prendendo come esempio il lavoro svolto precedentemente su MatLab [1] e migliorandolo in alcuni casi. Si partirà introducendo il problema da risolvere, facendo un esempio pratico, analizzando la possibile soluzione dal punto di vista matematico (quando possibile) ed infine commentando le parti di codice più significative.

6.3.1 Algoritmo di stima dell'angolo di assetto

L'angolo di assetto (β) è una grandezza fondamentale da stimare per valutare il comportamento di un veicolo in curva. Un calcolo preciso è ovviamente impraticabile utilizzando come strumentazione solamente uno smartphone, la via intrapresa è dunque quella di effettuare una *stima* nel modo più accurato possibile, raccogliendo i dati dai sensori e modellando il tutto con i filtri di *Kalman*.

In *dinamica del veicolo* [3] l'angolo di assetto è l'angolo compreso tra la direzione della velocità *assoluta* del veicolo, nel suo baricentro, e la direzione longitudinale della stessa.

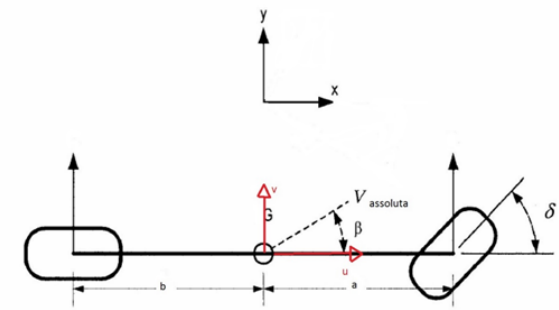


Figura 6.1. Rappresentazione approssimata dell'angolo di assetto.

Secondo tale definizione:

$$\beta = atan(\frac{v}{u}) \tag{6.1}$$

Per quanto riguarda la velocità *longitudinale* anche questa non può essere misurata direttamente poiché il valore restituito dal GPS non è propriamente interpretabile come componente longitudinale al veicolo ma è la somma dei vettori velocità *longitudinale* e velocità *laterale*. In genere l'angolo di assetto è comunque un angolo compreso tra 0°-3°, possiamo quindi approssimare la velocità *assoluta* misurata dal GPS come velocità longitudinale.

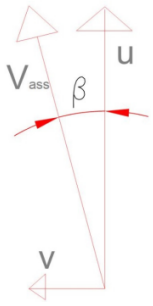


Figura 6.2. Schema vettoriale tra le velocità.

Per lo sviluppo dell'algoritmo si è deciso di utilizzare due filtri di *Kalman* in cascata; il primo **cinematico** che si occuperà di effettuare una stima del parametro u (*velocità longitudinale*) e che sarà utilizzato come ingresso per il secondo filtro **dinamico** che invece si occuperà di calcolare v (*velocità laterale*) e $\dot{\psi}$ (*velocità di imbardata*).

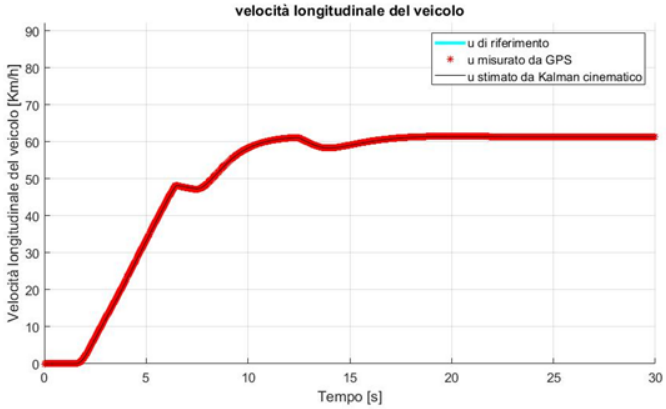


Figura 6.3. Velocità longitudinale ottenuta tramite applicazione filtro cinematico.

Per quanto riguarda l'applicazione del primo filtro notiamo come i risultati ottenuti siano molto coerenti con quanto aspettato dal modello teorico; per tale motivo la componente u può essere utilizzata come ingresso per il secondo filtro.

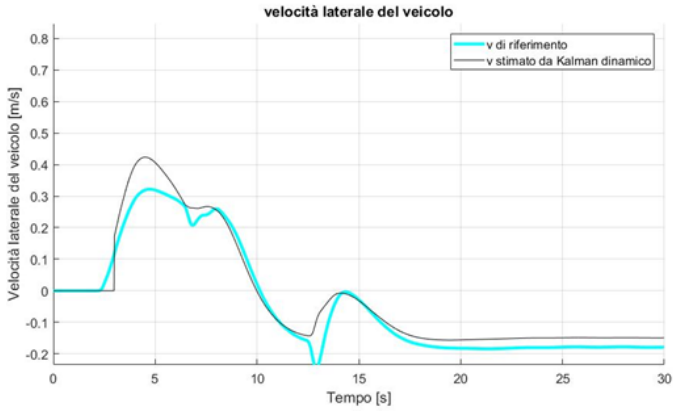


Figura 6.4. Velocità laterale ottenuta tramite applicazione filtro dinamico.

In questo caso la stima risulta meno *precisa* rispetto a u poiché è estrapolata interamente da un calcolo matematico tramite il filtro di $Kalman$ dinamico. Da notare comunque come l'approssimazione risulti comunque molto promettente su una misura esente da errori.

Quanto discusso è stato ovviamente implementato dapprima su MatLab [1] ottenendo dei risultati ottimi per quanto riguarda l'approssimazione dei valori (lo vedremo in seguito, nell'ultima parte dell'elaborato) peccando un po' in $performance$: tale è infatti stato l'obiettivo primario del lavoro svolto sul calcolo del β, rendere il tutto non elaborabile in *post* bensì svolgere il calcolo in *realtime*.

Introduzione alla stesura del codice

Inizialmente l'algoritmo è stato implementato *quasi* 1:1 con il codice MatLab precedentemente sviluppato ma il problema principale rimaneva il fatto che eseguire dei calcoli matematici tra matrici richiede una certa potenza computazionale (presente comunque sul processore del dispositivo utilizzato durante le prove [1]) ma sopratutto una ottimizzazione durante i calcoli. Tale ottimizzazione è *quasi* assente in ambiente Android o comunque non di certo ai livelli dei costrutti MatLab, pur utilizzando una libreria di notevole importanza nel panorama matematico su Java, **The Apache Commons Mathematics Library**. Uno degli obiettivi prefissati e raggiunti è stato quindi quello di ridurre i tempi di elaborazione durante il calcolo. Per far ciò, a differenza dell'algoritmo in MatLab, all'atto dell'acquisizione dei dati da parte dei sensori avviene una immediata memorizzazione dei valori in liste differenti per ogni tipo di dato acquisito; ogni lista viene *svuotata* da un thread differente che si occupa di mettere a disposizione il valore ad un thread *primario* che effettua i calcoli veri e propri.

[1] Specifiche tecniche del processore $Exynos8895$ equipaggiato su *Samsung Galaxy s8* `http://www.samsung.com/semiconductor/minisite/exynos/products/mobileprocessor/exynos-9-series-8895/`

Figura 6.5. Fase di acqusizione e smistamento dei dati.

I costrutti utilizzati per le *queues* saranno analizzati successivamente (6.4.3).

BetaCalculator class:

```
public class BetaCalculator {

    private Context context;
    private boolean execution = false;
    private LinkedBlockingQueue<GpsEntry> gpsQueue;
    private LinkedBlockingQueue<AccelEntry> accelQueue;
    private LinkedBlockingQueue<GyroEntry> gyroQueue;
    private LinkedBlockingQueue<StEntry> stQueue;

    private GpsEntry lastGpsEntry;
    private AccelEntry lastAccelEntry;
    private GyroEntry lastGyroEntry;
    private StEntry lastStEntry;
    private Activity activity;

    private RealMatrix X = MatrixUtils.createRealMatrix(2, 1);
    private RealMatrix Y = MatrixUtils.createRealMatrix(2, 1);


    private BroadcastReceiver mReceiver;
    private List<List<Double>> result = new ArrayList<>();
    private List<String> records = new ArrayList<>();
    private CarParameters params;
    ...

}
```

Come accennato poc'anzi la classe contiene delle *code* di dati organizzate come liste bloccanti per memorizzare tutti i parametri inviati dai sensori e ricevuti dal *Broadcast Receiver*

(5.1.3) il quale si occupa di inserirli ognuno nella propria lista di appartenenza. La chiamata al costruttore avviene nella *MainActivity* nel momento in cui viene avviata una prova ed il tutto avviene in un thread secondario:

```
...
BetaCalculator calculator = new BetaCalculator(this, (Activity) activity);

        threadCalculator = new Thread(new Runnable() {
            @Override
            public void run() {
                calculator.performCalculation();
            }
        });
        threadCalculator.start();
...
```

Il costruttore altro non fa che inizializzare le strutture dati ad una nuova prova, quindi evita che vi siano conflitti con prove precedentemente iniziate (per tale motivo non è stata utilizzata una classe statica). Il metodo $performCalculation()$ invece si occupa di *mettere in funzione* i thread riceventi i dati e questi cominciano ad estrarre dalla coda il dato significativo e memorizzarlo all'interno di una variabile globale di appoggio che verrà in seguito letta direttamente dall'algoritmo che si occupa di eseguire il calcolo. Questo aspetto è di fondamentale importanza poiché il meccanismo di calcolo in realtime in realtà tende ad utilizzare tutte le volte un *quartetto* di dati (velocità gps, angolo volante, accelerazione, velocità angolare) corrispondenti allo stesso istante di tempo; questo ovviamente non è sempre assicurato sopratutto perché il segnale GPS tarda ad arrivare (il tempo di campionamento è molto basso, si parla di circa un segnale ogni mezzo secondo); per cui la velocità viene considerata costante per tutto il tempo in cui il segnale del GPS tarda ad arrivare e. Per gli altri dati invece il meccanismo è un po' diversificato: i sensori di accelerometro e giroscopio lavorano entrambi alla stessa frequenza mentre il dato relativo all'angolo sterzo viene processato in ritardo; per cui i segnali di accelerazione e velocità angolare vengono man mano scartati fin quando non arriva un segnale utile dal *Service* dell'angolo volante e in quel momento i tre dati vengono affiancati al dato utile del GPS e vengono elaborati dal thread $Worker$.

I thread che raccolgono dati rimangono quindi in esecuzione per tutta la durata della prova e vengono *uccisi* contemporaneamente nel momento in cui la prova termina.

Di seguito una rapida occhiata al codice relativo alla parte successiva all'avvio dei thread secondari:

```
...
while (execution) {
            try {
                final double[][] var = executeFilters();
                if (var != null) {
                    if (result.isEmpty()) {
                        result = new ArrayList<>(var.length);
                        for (int i = 0; i < var.length; i++)
                            result.add(i, new ArrayList<Double>());
                    }
                    for (int i = 0; i < var.length; i++) {
                        result.get(i).add(var[i][0]);
                    }
                    if (positions.isEmpty()) {
                        positions = new ArrayList<>(2);
                        positions.add(new ArrayList<Double>());
                        positions.add(new ArrayList<Double>());
                    }
                    positions.get(0).add(lastGpsEntry.getLat());
                    positions.get(1).add(lastGpsEntry.getLon());
                    resetLastEntries();
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
}
```

La variabile *positions* invece serve a memorizzare i parametri di *Latitudine* e *Longitudine* raccolti dal GPS per tracciare un percorso approssimativo della prova. L'algoritmo vero e proprio invece è eseguito dalla chiamata:

```
final double[][] var = executeFilters();
```

che ritorna una *matrice* di dati contenente tutte le informazioni necessarie per costruire i grafici all'interno dell'activity *ShowResults*.

Il metodo verrà commentato per parti poiché risulta poco leggibile a primo impatto e ogni pezzo necessita di una spiegazione approfondita.

```
private double[][] executeFilters() {
...
RealMatrix a_s2_mat = MatrixUtils.createRowRealMatrix(accelerazioni);


        // rotazione SR
        RealMatrix a_s3_mat = v_A_s.multiply(a_s2_mat.transpose()).transpose();
        RealMatrix ax_n = a_s3_mat.getColumnMatrix(0);
```

```
        RealMatrix ay_n = a_s3_mat.getColumnMatrix(1);
        RealMatrix gyro_s2_mat = MatrixUtils.createRowRealMatrix(giroscopio);

        RealMatrix gyro_s3_mat = v_A_s.multiply(gyro_s2_mat.transpose())
        .transpose();
...
```

Come visto il metodo viene invocato ad ogni iterazione, quindi elabora più risultati che ritorna ad ogni ciclo. In questa prima parte vengono acquisiti i dati *grezzi* da elaborare, memorizzati in delle variabili aggiornate ad ogni ciclo e viene applicata, per le accelerazione e le velocità angolari (a_s2_mat, gyro_s2_mat) una *rotazione* alle componenti x, y, z di entrambe: ciò viene fatto poiché le misure acquisite dai sensori risentono della posizione del telefono all'interno del veicolo, in base a tale posizione e alla direzione del veicolo viene calcolata una matrice di rotazione (v_A_s), tramite una apposita fase di inizializzazione **??**), che viene utilizzata come *moltiplicatore* per tali misure.

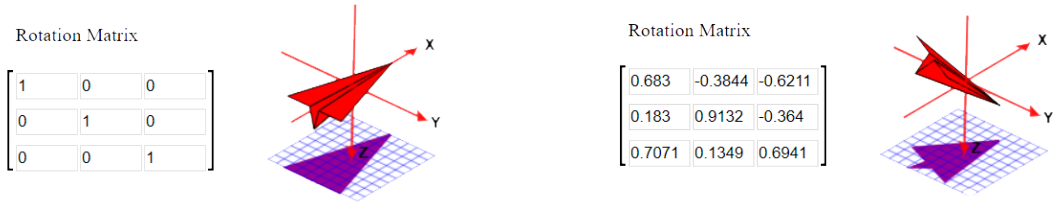


Figura 6.6. Applicazione di una rotazione tramite matrice di coseni.

```
...
double Ts_k = 0.001;
double Q_value = 10e-10;
double R_value = 10e-3;
RealMatrix G_st = MatrixUtils.createRealIdentityMatrix(2)
.scalarMultiply(Ts_k);
RealMatrix H_st = MatrixUtils.createRowRealMatrix(
new double[]{1, 0}
);
RealMatrix Q_st = MatrixUtils.createRealIdentityMatrix(2)
.scalarMultiply(10e-5);
double R_st = R_value;
RealMatrix P_st = Q_st.copy();
RealMatrix F = MatrixUtils.createRealMatrix(
new double[][]{{1, r_n * Ts_k},
{-r_n * Ts_k, 1}});
RealMatrix aux = MatrixUtils.createColumnRealMatrix(
new double[]{ax_n.getEntry(0, 0),
ay_n.getEntry(0, 0)});

kalmanfRealTime(X, F, G_st, aux, speed, H_st, P_st, Q_st, R_st);
...
```

In questo spezzone di codice viene applicato il primo dei due filtri in cascata di *Kalman*, in particolare quello cinematico: come spiegato in precedenza (3.1) il filtro restituisce in output, sotto forma di matrice X le due componenti di velocità u_n (velocità longitudinale) e v_n (velocità laterale); in ingresso invece riceve la stessa matrice X degli stati e delle matrici dipendenti da particolari parametri. I valori di Q ed R sono modificabili in base a:

- Q_value: più è basso il valore più affidabile sarà il *modello*.
- R_value: più è basso il valore più sarà affidabile la *misura*.

Ts_k rappresenta invece il tempo di campionamento. Le matrici Q_st e P_st sono le matrici di **covarianza** di modello e filtro. In questo primo filtro il valore di Q_st viene fissato a $10e-5$ poiché risulta molto più preciso il calcolo di u_n. Per quanto riguarda la funziona che applica il filtro:

```
private static void kalmanfRealTime(RealMatrix x, RealMatrix A, RealMatrix B,
                    RealMatrix u, double y, RealMatrix C,
                    RealMatrix P, RealMatrix Q, double R) {


        int n = x.getRowDimension();
```

Calcola il numero di stati del sistema (n) che nel caso del calcolo realtime sarà sempre 1;

```
        RealMatrix X_filtro = A.multiply(x.getColumnMatrix(0))
        .add(B.multiply(u));
```

il parametro X_filtro rappresenta invece lo stato del sistema in quell'istante seguendo la relazione

$$x = Ax + Bu \tag{6.2}$$

```
        P.setSubMatrix(A.multiply(P).multiply(A.transpose())
        .add(Q).getData(), 0, 0);
```

A questo punto viene modificata la matrice P secondo la relazione che lega gli errori con il modello:

$$P = APA' + Q \tag{6.3}$$

```
        RealMatrix a = P.multiply(C.transpose());
        RealMatrix b = C.multiply(P).multiply(C.transpose()).scalarAdd(R);
        b = new LUDecomposition(b).getSolver().getInverse();
        RealMatrix K = a.multiply(b);
```

Calcolo allora il guadagno di Kalman K come:

$$K = PC'(CPC' + R)^{-1} \tag{6.4}$$

```
        RealMatrix c = X_filtro.add(K.multiply(C.multiply(X_filtro)
        .scalarAdd(-y)).scalarMultiply(-1));
        X_filtro.setSubMatrix(c.getData(), 0, 0);
```

Avendo il guadagno posso calcolare il nuovo stato ottenuto come stato precedente sommato alla correzione:

$$x = x + K(y - Cx) \tag{6.5}$$

```
        if(n == 1) {
            RealMatrix d = MatrixUtils.createRealIdentityMatrix(n)
            .subtract(K.multiply(C));
            RealMatrix add1 = d.multiply(P).multiply(d.transpose());
            RealMatrix add2 = K.scalarMultiply(R).multiply(K.transpose());
            P.setSubMatrix(add1.add(add2).getData(), 0, 0);
        } else
            P.setSubMatrix(P.subtract(K.multiply(C).multiply(P))
            .getData(), 0, 0);
```

Stesso ragionamento applicato a P:

$$P = P - KCP \tag{6.6}$$

tale calcolo però avviene solamente per valori di $n > 1$, quindi nel nostro caso sarà invece calcolata P come:

$$P = [I - KC]P[I - KC]' + KRK' \tag{6.7}$$

```
        x.setColumnMatrix(0, X_filtro);
}
```

Infine memorizzo il nuovo stato nella matrice X e ritorno. Al termine del calcolo avrò quindi una matrice X del tipo

$$X = \begin{bmatrix} u_{nfiltrata} \\ v_{nfiltrata} \end{bmatrix} \tag{6.8}$$

Che rappresentano i due parametri sopra descritti $filtrati$ tramite il filtro cinematico. A questo punto si potrebbe dunque calcolare β ma i risultati non sarebbero lontanamente paragonabili a quelli ottenuti applicando il secondo filtro *dinamico*:

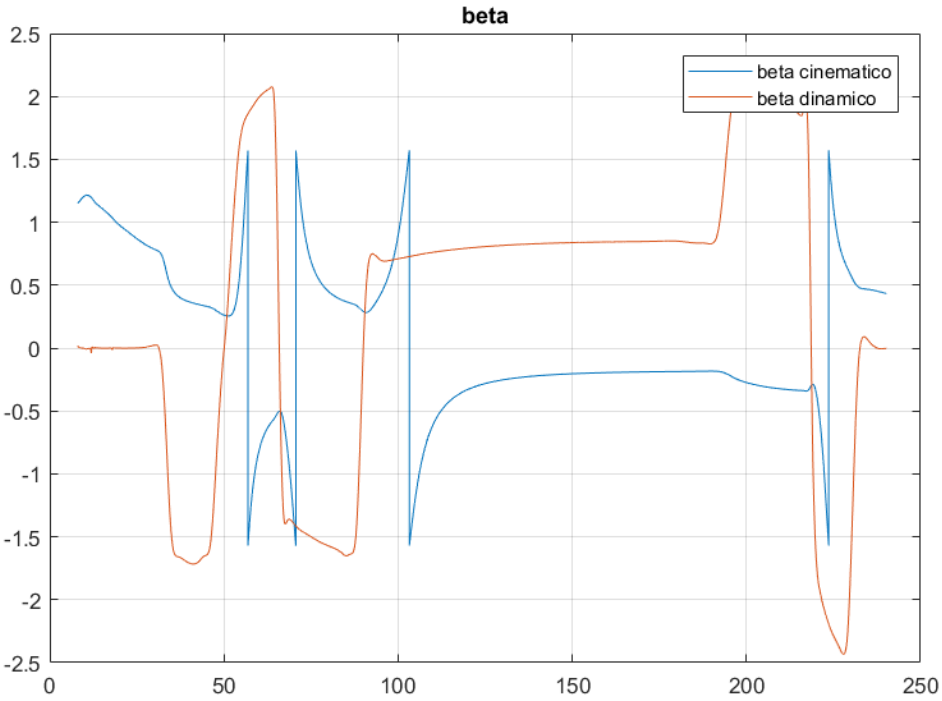


Figura 6.7. Prova simulata, differenza tra beta calcolato con il solo filtro cinematico e beta calcolato con i due filtri in cascata.

Il codice che segue riguarda l'implementazione del secondo filtro di *Kalman*:

```
double  u_n_filtrata = X.getEntry(0,0);
double  v_n_filtrata = X.getEntry(1,0);
double C1 = params.getC1(),
        C2 = params.getC2(),
        a1 = params.getA1(),
        a2 = params.getA2(),
        tau = params.getTau(),
        J = params.getJ(),
        m = params.getM();
V2 = ( -(C1 + C2) / (m * u_n_filtrata));
R2 = ( -((C1 * a1 - C2 * a2) / (m * u_n_filtrata) + u_n_filtrata));
U2 = ( (C1 * v_0 + C1 * r_0 * a1 - C2 * a2 * r_0 + C2 * v_0)
/ (m * Math.pow(u_n_filtrata, 2)) - r_0);
D2 = ( C1 * tau / m);
V3 = ( -(C1 * a1 - C2 * a2) / (J * u_n_filtrata));
R3 = ( -(C1 * Math.pow(a1, 2) + C2 * Math.pow(a2, 2))
/ (J * u_n_filtrata));
U3 = ( (C1 * a1 * v_0 + C1 * r_0 * Math.pow(a1, 2) +
 C2 * Math.pow(a2, 2) * r_0 - C2 * a2 * v_0)
```

```
 / (J * Math.pow(u_n_filtrata, 2)));
D3 = ( C1 * tau * a1 / J);
```

I valori di u_n e v_n, estrapolati dalla matrice risultato X vengono utilizzati come input per il secondo filtro insieme ai parametri riguardanti la classificazione dei veicoli in segmenti (6.4.1) e delle matrici dipendenti dai parametri appena descritti.

```
RealMatrix C_dyn = MatrixUtils.createRowRealMatrix(
new double[]{0, 1});
RealMatrix Q_dyn = MatrixUtils.createRealMatrix(new double[][]{
{Q_value, 0}, {0, Q_value}});
RealMatrix P_dyn = Q_dyn.copy();
double R_dyn = R_value;
F = MatrixUtils.createRealMatrix(new double[][]{
        {1 + Ts_k * V2, Ts_k * R2},
        {Ts_k * V3, (1 + Ts_k * R3)}
});

RealMatrix G_bis = MatrixUtils.createRealMatrix(new double[][]{
        {Ts_k * U2, Ts_k * D2},
        {Ts_k * U3, Ts_k * D3}});

aux = MatrixUtils.createColumnRealMatrix(new double[]{
u_n_filtrata, volante.getTheta()});

kalmanfRealTime(Y, F, G_bis, aux, r_n, C_dyn, P_dyn, Q_dyn, R_dyn);
```

Come descritto per il filtro *cinematico* anche in questo caso si vanno ad utilizzare delle matrici di covarianza (P e Q) e delle matrici di appoggio dipendenti dai valori u_n, v_n e dai parametri del segmento di veicolo scelto. Il filtraggio avviene allo stesso modo del filtro cinematico, solamente che la matrice risultante Y sarà del tipo:

$$Y = \begin{bmatrix} v_{nfiltrata} \\ r_{nfiltrata} \end{bmatrix} \tag{6.9}$$

A questo punto è possibile calcolare β (angolo di assetto come:

$$\beta = atan(v_{nfiltrata}/u_{nfiltrata}) \tag{6.10}$$

Successivamente viene creata una stringa contenente tutti i dati utili a ricostruire i grafici per la misurazione ed inserita in una lista apposita contenente tutti i campioni per una singola prova:

```
StringBuilder sb = new StringBuilder();
...
sb.append(r_n);
sb.append(";");

// Vel. longitudinale
sb.append(u_n_filtrata);
```

```
sb.append(";");

// Volante
sb.append(volante.getTheta());
sb.append(";");

// Beta
sb.append(beta_dinamico[0]);
sb.append(";");


records.add(sb.toString());
```

Tale lista servirà nel momento in cui l'utente voglia salvare i dati grezzi in un file .CSV ed inviarli per e-mail. Infine viene ritornata una matrice contenente tutti i valori.

```
return new double[][]{beta_dinamico,
    a_s2_mat.getColumn(0),
    a_s2_mat.getColumn(1),
    a_s2_mat.getColumn(2),
    {volante.getTheta()},
    gyro_s2_mat.getColumn(0),
    gyro_s2_mat.getColumn(1),
    gyro_s2_mat.getColumn(2),
    {u_n_filtrata, r_n}};
```

Come accennato inizialmente il metodo continuerà a girare fin quando l'utente non termina la prova, quindi collezionerà circa un *migliaio* di record ogni venti secondi. Ovviamente la mole di dati da spostare e calcolare, anche nel momento in cui la prova voglia essere rivista in futuro, è enorme. Per tale motivo l'algoritmo inizialmente sviluppato su MatLab [1] è stato ripensato e modificato per cominciare ad elaborare non appena la prova cominci e non svolgere tutto il calcolo in post.

Modifiche e miglioramenti apportati all'algoritmo

Come accennato l'algoritmo precedentemente sviluppato accumulava i dati grezzi necessari, costruiva le strutture dati necessarie per contenere la mole di dati ed applicava, in modo iterativo, i filtri ed otteneva, per ogni stato del sistema, i due valori appena visti. In questo modo però il tempo di esecuzione è ovviamente *dipendente* dal numero di record raccolti. La prima miglioria apportata a tale approccio è stata l'implementazione di un algoritmo che lavorasse allo stesso modo ma iniziasse l'elaborazione fin da subito: ragionando sul fatto che l'unica dipendenza che ha il calcolo è lo stato *immediatamente* precedente sia per X che per Y si è pensato allora di mantenere la struttura a matrice ma di limitare la grandezza ad una matrice 2x1. I risultati ottenuto hanno giovato molto sul fronte $per formance$ come si evince dal grafico in basso:

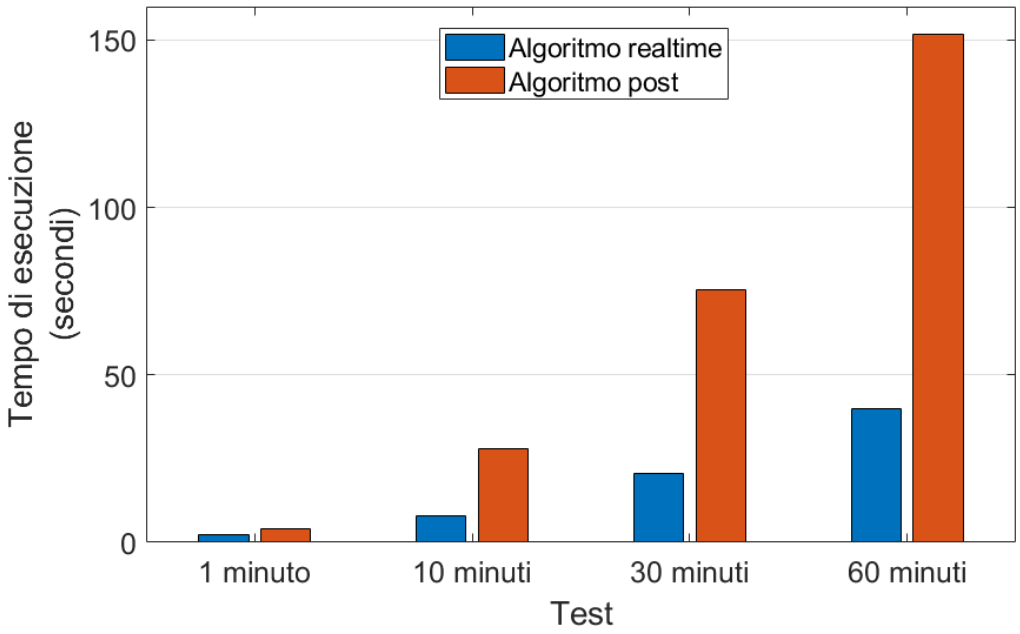


Figura 6.8. Confronto tra il porting dell'algoritmo da MatLab (*Post*) e lo stesso algoritmo ottimizzato in RealTime. (*Realtime*)

Il fatto che il tempo di esecuzione dell'algoritmo in realtime non rasenti lo zero è dovuto al fatto che nel caso di grosse moli di dati (per la prova di 30 minuti i dati raccolti sono circa 105.000, per la prova di 60 minuti circa 220.000) vi è un *overhead* marginale: l'algoritmo non riesce a processare i dati in un tempo minore o uguale al tempo di *campionamento* che corrisponde a circa 0.015 secondi. Accumulando tale *overhead* man mano che i records aumentano, aumenta anche il tempo di attesa dopo ogni prova. Tale tempo risulta essere però *molto* minore rispetto ad una esecuzione in *post*: il rapporto è di circa 1:4. Per adattare l'algoritmo ad una esecuzione di questo tipo è stato manipolato leggermente il codice e le strutture dati; avendo una dipendenza solamente dallo stato precedente il calcolo del β, che risulta comunque essere una stima, è *molto* più vicino al dato teorico o comunque del modello. Dal grafico in basso si nota come l'algoritmo in *realtime* riesca ad ottenere risultati *molto* più promettenti dell'algoritmo precedentemente sviluppato in MatLab.

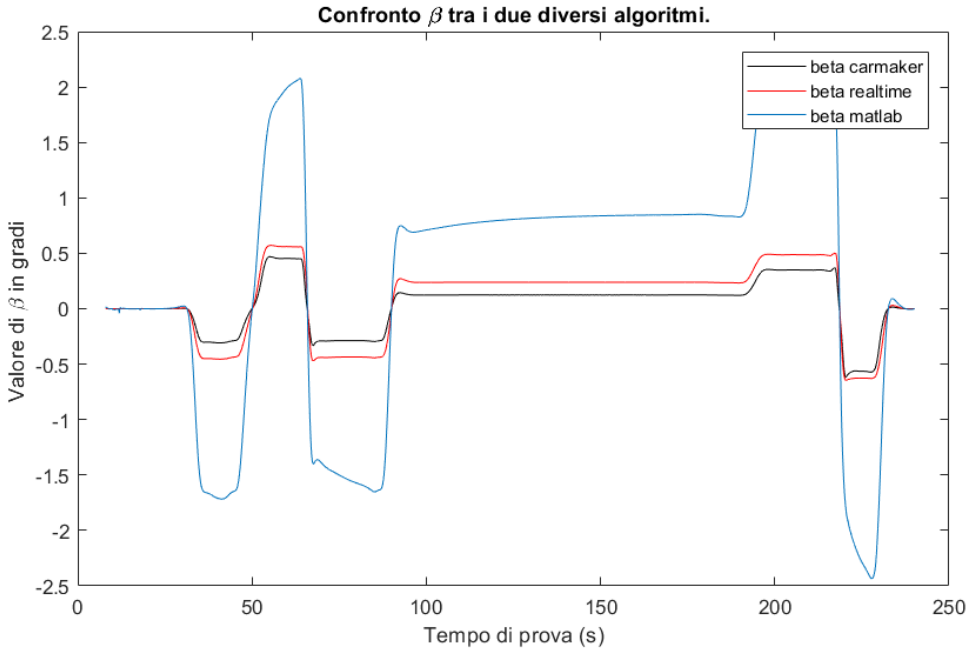


Figura 6.9. Confronto dei risultati ottenuti da una prova eseguita sul simulatore CarMaker.

I dati in input sono stati ottenuti tramite un simulatore, *CarMaker* e posti in input ad entrambi gli algoritmi: quello scritto in MatLab e quindi *runnato* direttamente in ambiente Windows e il porting dello stesso su Android e modificato per elaborare in realtime. I risultati sono molto promettenti, si nota infatti come la curva *nera*, valore calcolato direttamente dal software di simulazione, sia molto più simile a quella *rossa*, risultati ottenuti su Android, rispetto a quella *blu*, codice eseguito su MatLab.

6.3.2 Algoritmo di calcolo del sistema di riferimento veicolo-smartphone.

Introduzione

Nel corso della trattazione, anche se non specificato, sono stati utilizzati più sistemi di riferimento e non solo quello utilizzato dal dispositivo:

- Sistema di riferimento **assoluto** (X_0, Y_0, Z_0) fisso: è utile per riportare le grandezze assolute quali l'accelerazione di gravità ed il campo magnetico terrestre.
- Sistema di riferimento **veicolo** (X_V, Y_V, Z_V) con centro nel baricentro del veicolo : tale sistema è mobile ma non ruota; permette allora di calcolare gli angoli di rotazione del veicolo durante una sterzata.
- Sistema di riferimento **smartphone** (X_S, Y_S, Z_S): tutte le grandezze misurate dai sensori sono riferite a questo sistema di riferimento.

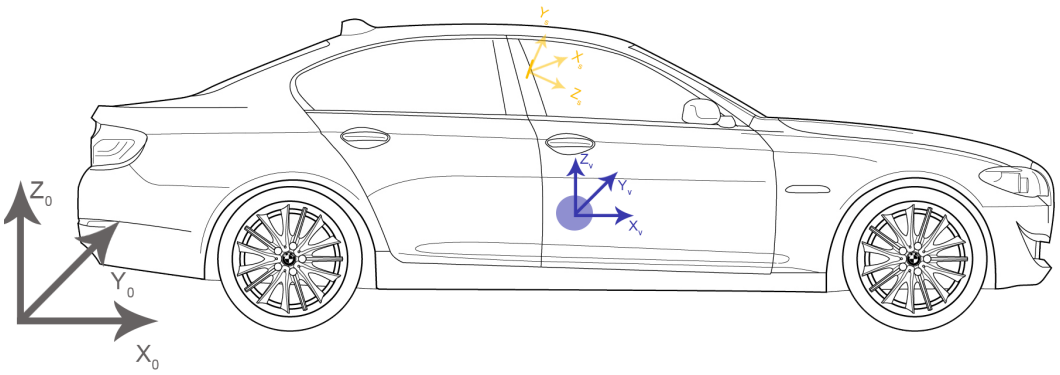


Figura 6.10. Vista laterale dei sistemi di riferimento.

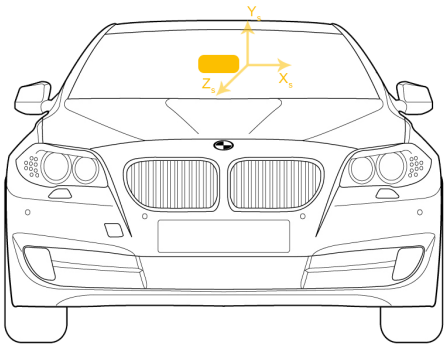


Figura 6.11. Vista frontale del sistema di riferimento smartphone.

Come detto tutte le grandezze vengono misurate nel sistema di riferimento smartphone (X_S, Y_S, Z_S) ma ovviamente nella modellazione dei sistemi dinamici abbiamo bisogna delle

misure nel sistema di riferimento veicolo (X_V, Y_V, Z_V), come ad esempio le misure delle accelerazioni e delle velocità angolari, devono essere misurate in relazione al baricentro del veicolo. Per far ciò è stata creata una classe apposita che si occupa di svolgere la fase di *inizializzazione* delle misure: prima di ogni prova, nel caso in cui l'utente lo stabilisse, è possibile svolgere tale fase in cui bisogna effettuare delle misurazioni in condizioni specifiche. Verrà infatti chiesto all'utente di mantenere la vettura immobile per poi cominciare ad accelerare in un tratto rettilineo: questo è utile per capire la condizione iniziale dei sensori e quindi come è posizionato lo smartphone all'interno del veicolo e, con la seconda fase, capire la direzione longitudinale di manovra del veicolo stesso. Tramite tali misurazioni è possibile estrapolare, con un algoritmo apposito, la **matrice di rotazione** relativa alla posizione dello smartphone rispetto al veicolo. La relazione che lega le misure al sistema di riferimento è data da:

$$\begin{pmatrix} r_{xb} \\ r_{yb} \\ r_{zb} \end{pmatrix} = \begin{bmatrix} c_{xs-xb} & c_{xs-yb} & c_{xs-zb} \\ c_{ys-xb} & c_{ys-yb} & c_{ys-zb} \\ c_{zs-xb} & c_{zs-yb} & c_{zs-zb} \end{bmatrix} \begin{pmatrix} r_{xs} \\ r_{ys} \\ r_{zs} \end{pmatrix} \tag{6.11}$$

in cui:

- $\begin{pmatrix} r_{xb} \\ r_{yb} \\ r_{zb} \end{pmatrix}$ rappresenta una generica misura r riportata agli assi dei veicolo (X_V, Y_V, Z_V).
- La matrice $\begin{bmatrix} c_{xs-xb} & c_{xs-yb} & c_{xs-zb} \\ c_{ys-xb} & c_{ys-yb} & c_{ys-zb} \\ c_{zs-xb} & c_{zs-yb} & c_{zs-zb} \end{bmatrix}$ è la matrice dei *coseni direttori* che determinano la rotazione del sistema di riferimento smartphone rispetto a quello veicolo. (il termine c_{xs-xb} ad esempio rappresenta il coseno dell'angolo tra l'asse dello smartphone X_S e quello del veicolo X_V).
- $\begin{pmatrix} r_{xs} \\ r_{ys} \\ r_{zs} \end{pmatrix}$ rappresenta invece la misurazione della grandezza r nel sistema di riferimento smartphone.

Tramite questa relazione è possibile trasformare ogni grandezza ottenuta nel sistema di riferimento dei sensori in un altro, invertendo la matrice dei coseni è possibile ottenere l'inverso, ossia passare dal sistema di riferimento veicolo a quello sensori.

Codice di riferimento su MatLab

Come accennato i dati significativi utilizzati per poter estrapolare la matrice di rotazione che rappresenta le misurazioni nel sistema di riferimento sensori, vengono estratti da una fase di inizializzazione (facoltativa nel caso in cui l'utente voglia mantenere la stessa configurazione nel corso del tempo) che consta di due fasi intermedie. Nella descrizione seguente si prenderà come esempio un dispositivo ruotato come segue:

- x: 20 gradi;
- y: 40 gradi.
- z: 30 gradi.

- **Veicolo fermo**, in cui vengono raccolti i dati relativi ad accelerazioni e velocità angolari rispettivamente da accelerometro e giroscopio ed in base alle componenti si riesce a capire come il telefono è posto rispetto al sistema di riferimento terrestre. Analizzando i dati in MatLab otteniamo un grafico del tipo:

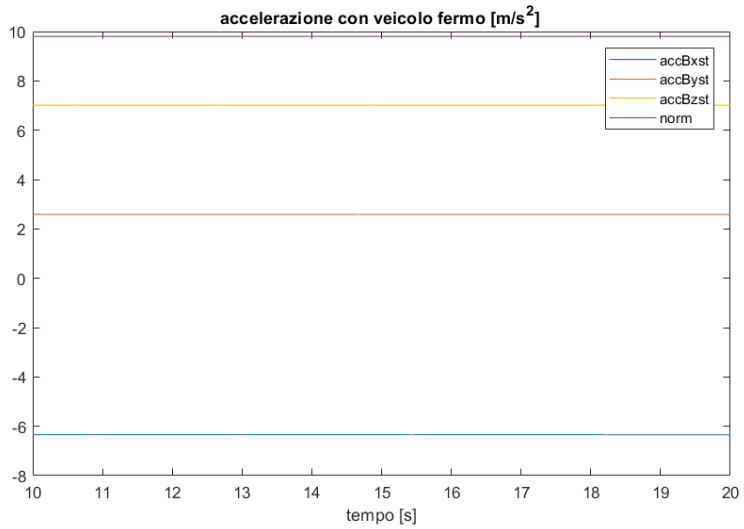


Figura 6.12. Grafico delle accelerazioni rispetto al sistema di riferimento sensori a veicolo fermo.

Notiamo come il valore di g sia ripartito nelle tre componenti x, y, z delle accelerazioni.

- **Veicolo in accelerazione**, in questa fase le componenti delle accelerazioni variano a seconda di come il veicolo sta accelerando e verso quale direzione ed anche in funzione dei cambi marcia che provocano un picco negativo seguito da uno positivo.

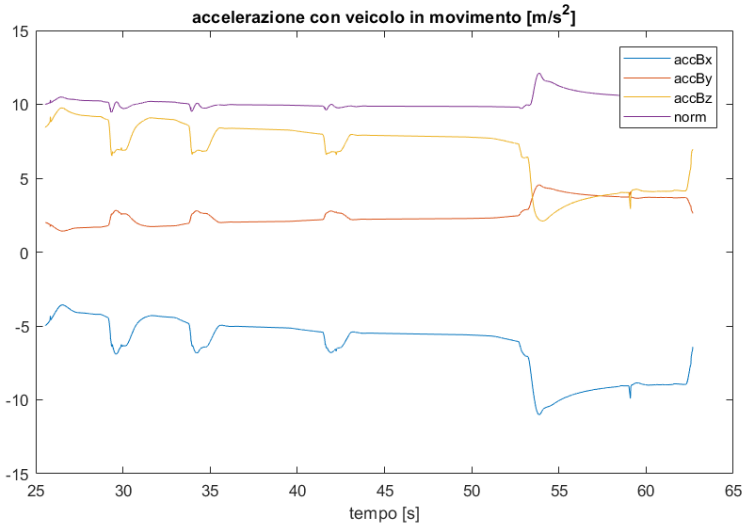


Figura 6.13. Grafico delle accelerazioni rispetto al sistema di riferimento sensori con veicolo in accelerazione.

Partendo da questi due grafici ed in particolare quello in cui il veicolo è in accelerazione notiamo che, ripartendo il grafico in tre sotto-grafici, per ogni componente, come quello che si viene a formare è una *nuvola* di punti che dapprima si sviluppa in senso positivo (veicolo in accelerazione) e poi si riversa su se stessa nel momento in cui il veicolo frena.

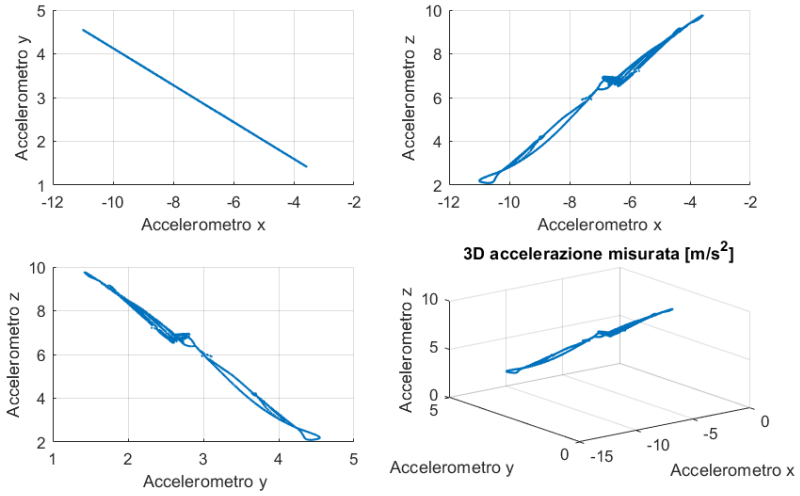


Figura 6.14. Grafico delle accelerazioni rispetto al sistema di riferimento sensori rispetto alle tre componenti e in 3D.

L'obiettivo adesso è quello di trovare un *piano* che intersechi i punti trovati, tramite

interpolazione: in questo modo si può calcolare in seguito il *versore* perpendicolare al piano e di conseguenza i tre versori che rappresentano il sistema di riferimento del veicolo in prova.

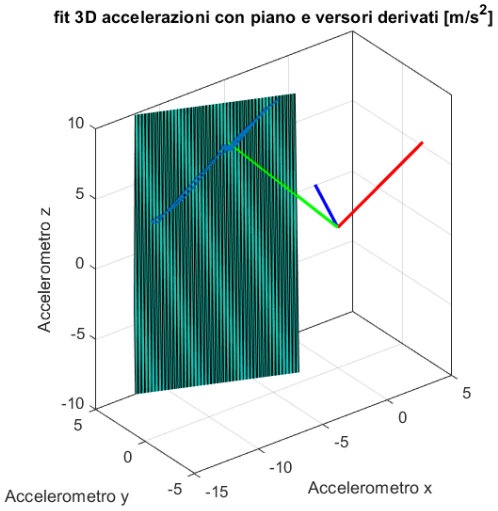


Figura 6.15. Il grafico identifica un piano passante per i punti trovati e i versori derivati dal piano

Il piano viene identificato utilizzando la funzione fit in MatLab che restituisce, dati un insieme di punti, i coefficienti di un piano (a, b, c) passante per quei punti e (nel caso in esame) passante per zero. I versori vengono poi identificati tramite prodotti vettoriali successive del versore g, inizialmente con il versore y identificato dal vettore perpendicolare al piano, per calcolare x. Successivamente z viene calcolato dal prodotto vettoriale di x con y. Partendo da questo esempio (gentilmente elaborato dal Prof. Pastorelli del Politecnico di Torino) si può sviluppare un algoritmo simile sullo smartphone utilizzando librerie opportune e calcoli similari.

FitHelper Class:

```
public class FitHelper {
    static BroadcastReceiver stReceiver;
    static BroadcastReceiver dynReceiver;
    static List<Point3D_F32> stData = new ArrayList<>();
    static List<Point3D_F32> dynData = new ArrayList<>();
    static Context ctx;
    static DescriptiveStatistics meanValue;
    ...
```

La classe presenta degli oggetti e metodi *statici* per essere semplicemente richiamata senza alcuna istanza; i due $BroadCastReceiver$ saranno utilizzati per ricevere, dai sensori, i valori di accelerazione quando il veicolo è fermo (fase 1) e quando è in accelerazione (fase 2). Ovviamente non lavoreranno in contemporanea. Le due liste di $Point3D_F32$, che

rappresenta uno standard per la libreria utilizzata [20] atto alla memorizzazione di punti sulla spazio in Floating Point a 32 bit, andranno a memorizzare rispettivamente i dati durante la fase *statica* (a veicolo fermo) e durante la fase *dinamica* (a veicolo in moto). Infine il valore $meanValue$ rappresenta un costrutto utile ad immagazzinare dati all'interno, specificando a priori la dimensione massima della *window* e successivamente, tramite una semplice chiamata, calcolare vari tipi di parametri statistici tra cui la media.

```
public static void init (Context context, boolean storedValues,
                         final int phase) {
        ctx = context;
        meanValue = new DescriptiveStatistics(50000);
        if(storedValues) {
            StorageUtility.readFileAccs(stData, dynData);
        } else {
        ...
```

Il metodo *init* è appunto un metodo richiamato prima di effettuare il calcolo matematico, è infatti utile per inizializzare i due *receiver* e contiene un parametro $storedValues$ che permette di specificare se effettuare la calibrazione *manualmente* o leggere da file una calibrazione precedente ed evitare quindi di effettuarla ad ogni test. Successivamente, se bisogna effettuare la calibrazione vengono istanziati i due *receiver* a seconda della fase corrente (parametro *phase*):

```
        if (phase == 1 && stReceiver == null) {
               stReceiver = new BroadcastReceiver() {
                   @Override
                   public void onReceive(Context context, Intent intent) {
                   ...
             }
        else if(phase == 2 && dynReceiver == null) {
               dynReceiver = new BroadcastReceiver() {
                   @Override
                   public void onReceive(Context context, Intent intent) {
                   ...
```

I due metodi successivi saranno utili invece per la fase terminale dell'algoritmo in quanto:

```
public static void destroy() {
        if(ctx == null || stReceiver == null) return;
        ctx.unregisterReceiver(stReceiver);
        stReceiver = null;
    }
```

Permette di, una volta terminata la $fase$ 1, di eliminare il *receiver* associato ed inizializzarlo per la prossima volta che sarà utilizzato.

```
public static void saveResults() {
        if(ctx == null || dynReceiver == null) return;
        ctx.unregisterReceiver(dynReceiver);
        dynReceiver = null;
        AsyncTask.execute(new Runnable() {
            @Override
            public void run() {
                StorageUtility.writeFileAccs(stData, dynData);
                meanValue.clear();

            }
        });
    }
```

Permette invece di, una volta terminata la *fase* 2, di eliminare il *receiver* associato e scrivere su un file i risultati che potranno poi essere utilizzati in futuro, se si volesse riutilizzare la stessa configurazione per la prova.

```
public static RealMatrix retriveRotationMatrix(Context context)
throws ExecutionException, InterruptedException {
        return new execute(context).executeOnExecutor(
        AsyncTask.THREAD_POOL_EXECUTOR
        ).get();
    }
```

Tale metodo rappresenta la chiamata vera e propria al calcolo della matrice dei coseni direttori; non fa altro che eseguire un task in background, utilizzando il costrutto *AsyncTask*, che esegue il calcolo e *ritorna*, come valore, un oggetto *RealMatrix*. Il costrutto utilizzato permette, nel momento in cui il metodo viene invocato, di *aspettare* che il metodo elabori e ritorni, nonostante il calcolo sia fatto in background. Tale costrutto è noto come *costrutto bloccante* e permette di eseguire comunque un task in background ma di *attendere* il risultato nel thread chiamante il metodo.

```
private static class execute extends
AsyncTask<Void, Void, RealMatrix> {

        Context context;
        RealMatrix rot;
        RealMatrix fact;
        FitPlane3D_F32 f32;
```

La classe *execute* contiene delle matrici di appoggio *rot* e *fact* utilizzate, come nel codice MatLab, per il calcolo della matrice di rotazione e un oggetto di tipo **FitPlane3D_F32** che permette di richiamare dei metodi *statici* di calcolo di un piano dati un set di punti; in particolare la funzione utilizzata:

```
        f32.svd(stData, ver_gIMU, ver_y);
        ver_gIMU.divideIP(ver_gIMU.norm());
        f32.svdPoint(dynData, new Point3D_F32(0, 0, 0), ver_y);
```

Viene dapprima chiamato il metodo *svd* che, passato il set di punti *statico* permette di calcolare il *centroide* del piano, identificato come vettore che nel nostro caso sarà rappresentato come il vettore g e successivamente normalizzato per ottenere il versore corrispondente. La seconda chiamata al metodo *svdPoint* è molto simile alla prima, solamente che viene passato come primo parametro il set di punti *dinamico*, come secondo parametro un punto generico appartenente al piano (viene passato l'origine proprio perché in MatLab è una condizione necessaria che il piano passi per l'origine) e ritorna, come terzo parametro, un vettore *normale* al piano e centrato nel punto passato come secondo parametro. Ciò permette di identificare il versore y e quindi tramite prodotto vettoriale di ottenere:

```
ver_x = ver_y.cross(ver_gIMU.toVector());
ver_z = ver_x.cross(ver_y);
```

Il versore x e successivamente ricalcolare il versore z, corrispondente a g solamente come ulteriore prova di quanto fatto. Viene allora calcolata una prima matrice di rotazione dai valori corrispondenti ai versori:

```
rot.setEntry(0,0,ver_x.getX());
rot.setEntry(0,1,ver_y.getX());
rot.setEntry(0,2,ver_z.getX());

rot.setEntry(1,0,ver_x.getY());
rot.setEntry(1,1,ver_y.getY());
rot.setEntry(1,2,ver_z.getY());

rot.setEntry(2,0,ver_x.getZ());
rot.setEntry(2,1,ver_y.getZ());
rot.setEntry(2,2,ver_z.getZ());

rot = rot.transpose();
```

Che dovrà essere opportunamente modificata in base al valore della media calcolata sui punti durante l'acquisizione della parte *dinamica*: tale procedimento permette infatti di non avere errori dovuti al fatto che un angolo, ad esempio, di 45 gradi possa essere interpretato come un angolo di 135 gradi, a seconda di come viene considerata la rotazione; seguendo tale ragionamento la media mi restituirà un valore positivo o negativo a seconda del verso in cui l'accelerazione va misurata in fase di movimento rispetto alla lettura effettuata invece dall'accelerometro:

```
double mean = meanValue.getMean();
if(mean < 0) {
    ver_x.timesIP(-1);
    ver_y.timesIP(-1);

    rot.setEntry(0,0,ver_x.getX());
    rot.setEntry(0,1,ver_y.getX());
    rot.setEntry(0,2,ver_z.getX());

    rot.setEntry(1,0,ver_x.getY());
    rot.setEntry(1,1,ver_y.getY());
```

```
            rot.setEntry(1,2,ver_z.getY());

            rot.setEntry(2,0,ver_x.getZ());
            rot.setEntry(2,1,ver_y.getZ());
            rot.setEntry(2,2,ver_z.getZ());

            rot = rot.transpose();
        }
```

In tal caso infatti viene ricalcolata la matrice con i segni invertiti sia su x che su y, poiché siamo sicuri che la misurazione stava avvenendo nel verso opposto rispetto a quello che andrà considerato. Infine viene ritornata la matrice:

```
        return rot;
    }
```

Tale algoritmo è utilizzato, come abbiamo visto (6.3.1) per comporre in maniera corretta la matrice di rotazione nel calcolo della stima del β.

6.3.3 Algoritmo di stima dell'angolo volante.

Come visto precedentemente l'angolo volante (θ) rappresenta l'ultimo dei parametri necessari per calcolare il β; calcolarlo tramite gli appositi sensori della rete CAN del veicolo non è molto complesso (a patto di avere l'autorizzazione per poterlo fare), mentre utilizzare un metodo non *tradizionale* è risultato molto più problematico.

L'approccio al problema

Ciò che contraddistingue l'algoritmo di calcolo dell'angolo volante dagli altri *macro* algoritmi è il fatto che è stato ideato praticamente da zero: come già visto (4) si è utilizzata una libreria apposita di *Computer Vision* per realizzare l'algoritmo che elabora le immagini tramite trasformazioni applicate su matrici (di pixel). L'idea che sta alla base è quella di non dover intaccare in alcun modo la rete *interna* del veicolo(CAN, centralina) e di poter essere il più economicamente possibile sostenibile e priva di vincoli dovuti a particolari strumentazioni: basta infatti stampare dei *markers* da applicare su un cartoncino nero per realizzare il tutto.

Figura 6.16. Applicazione sul volante della "strumentazione" necessaria.

Il problema si riduce allora a dover utilizzare la camera del dispositivo per poter individuare e tracciare i markers sul volante: ovviamente è metodo non esente da difetti e problemi di vario tipo, tra cui:

- **Pessime condizioni di luce**, l'algoritmo lavora su una proiezione in B/W dell'immagine, un calo repentino di luminosità o un aumento provoca una fase *cieca* di ricerca dei markers; molto probabilmente si finisce per perderli del tutto.
- **Eccessive vibrazioni**, essendo montato in auto è comunque sensibile alle vibrazioni provocate dall'auto stessa; nonostante implementi una funzione di $follow$ dei punti se la vibrazione è molto accentuata rischia che l'immagine si trovi fuori dall'angolo di acquisizione della camera.
- **Ostacoli tra camera e volante**, basti pensare alle braccia del guidatore: in caso di manovra azzardata o di curva con ampio raggio il pilota può in certi casi coprire i markers con le braccia involontariamente. Nonostante l'algoritmo implementi una funzione di *recovery* dei punti, se si perde più di un punto in un singolo frame o comunque un punto per più frame il calcolo è pesantemente compromesso.

La logica dell'algoritmo è stata sviluppata interamente all'interno di tale classe pur non essendo la classe di riferimento per l'interfacciamento alla camera che avviene, come vedremo (6.4.3, **??**) in due parti distinte. L'introduzione al codice risulta comunque molto complessa poiché la classe contiene un numero di variabili e metodi molto elevato a causa della complessità che vi è all'interno; inoltre bisogna introdurre anche le classi *secondarie* legate alla classe $ObjectTracker$ utilizzate rispettivamente per rappresentare l'oggetto generico $Marker$ e per effettuare il calcolo in realtime dell'angolo volante, dati due set di punti. L'approccio intrapreso per spiegare al meglio la procedura è di tipo $top - down$: si partirà dalla chiamata al metodo e successivamente si analizzerà il flusso di funzionamento contornato appunto dalle due classi secondarie.

Funzionamento generale dell'algoritmo

L'algoritmo può essere diviso in due macro *rami* principali di funzionamento:

- **Fase di Individuazione**,

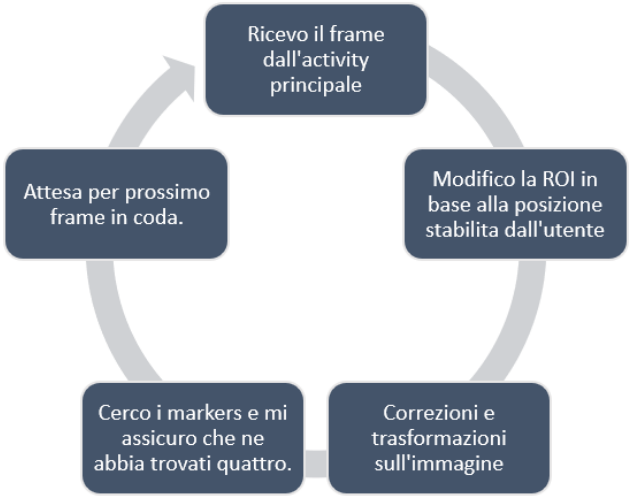


Figura 6.17. Ciclo di funzionamento generale della prima fase di esecuzione dell'algoritmo.

in questo frangente l'agoritmo non fa altro che ricercare i marker all'interno del frame senza applicare filtri di ricerca o qualche logica; individua, in base alle trasformazioni sull'immagine e ai filtri applicati ad essa, le parti *bianche* e ne calcola il centroide individuandolo come punto. E' compito dell'utente prestare attenzione e vedere quando effettivamente tutti e quattro i marker si trovano all'interno della *ROI* e premere il pulsante apposito per cominciare la seconda fase di analisi. Nel momento in cui ciò viene fatto l'algoritmo salva in locale le coordinate dei punti individuati che utilizzerà, come posizione di riferimento iniziale dei marker. Inoltre viene calcolata una distanza media tra i punti semplicemente ciclando sulla lista di marker individuati, per stabilire una distanza minima rispettabile nel momento in cui vi è il calcolo della distanza tra i marker di due frame successivi.

- **Fase di Calcolo**,

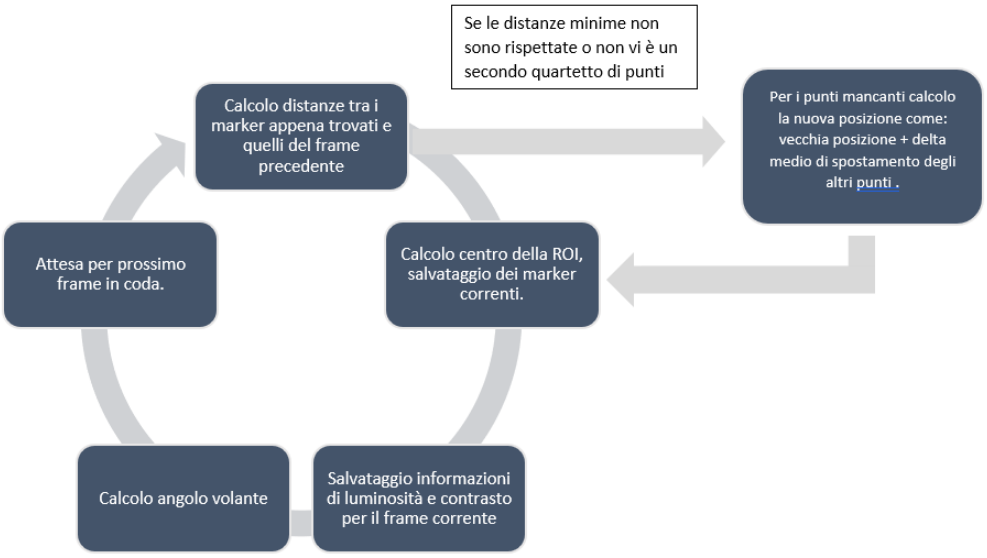


Figura 6.18. Ciclo di funzionamento generale dell'algoritmo..

La fase di *Calcolo* o *Registrazione* consta di quattro fasi principali di funzionamento; c'è da considerare il fatto che le due fasi non sono totalmente slegate anzi, la fase corrente riprende sempre come base il funzionamento della fase di individuazione poiché la ricerca dei marker all'interno del frame presuppone l'esecuzione di quella parte di codice:

– **Fase 1**, la prima operazione effettuata è il calcolo delle distanze per ogni marker individuato, rispetto a tutti i marker individuati nel frame precedente, e ognuna viene prima confrontata con un valore minimo di distanza (calcolato nel momento in cui viene avviata la fase di calcolo).

– **Fase 2**, può essere eseguita in due diversi modi a seconda di quello che accade nella prima fase: se i marker vengono immediatamente associati a quelli del frame precedente senza avere alcuna incertezza durante tale fase viene solamente spostato il centroide della *ROI*, e salvate le informazioni riguardanti le coordinate dei nuovi marker individuati. Altrimenti avviene una fase di *recovery*: per ogni marker non individuato o perso viene effettuata una *stima* di posizionamento in base alla posizione nel frame precedente e allo spostamento medio degli altri marker. Ogni struttura infatti memorizza un valore, *delta*, che indica il verso (positivo o negativo) di spostamento e il valore di tale spostamento; operando con una semplice media si stima quindi la rotazione dei punti che non vengono inizialmente riconosciuti dall'algoritmo.

– **Fase 3**, il frame corrente è adesso considerato come *valido*, per cui sono sicuro che in tale situazione i marker sono visibili o comunque stimabili dall'algoritmo: vengono allora salvate le informazioni riguardanti le variazioni di luminosità e contrasto del frame rispetto al precedente per mantenere un riferimento da seguire nel caso non fossero visibili successivamente.

– **Fase 4**, nella fase finale avviene il calcolo vero e proprio della rotazione a partire dalle coordinate dei marker nel frame precedente e la variazione di spazio nel frame corrente: ciò, come vedremo in seguito, avviene utilizzando l'algoritmo specifico di *Procrustes*.

ObjectTracker Class:

```
public class ObjectTracker {

    private static final int N_POINTS = 4;
    private static double initContrast;
    private static double initBrightness;
    private static Point mRoiCenter;
    private static double theta = 0;
    ...
    private static boolean isRecording = false;
    private static List<Tracker> currFramePoints = new ArrayList<>();
    private static List<Tracker> prevFramePoints = new ArrayList();
    private static MatOfPoint mContour;
    private static SteeringUtilities st;
    private static ArrayList<Tracker> initPoints;
```

Come anticipato la classe è abbastanza corposa, saranno dunque omessi sia certi metodi, considerati *secondari* sia certe variabili. Dal primo spezzone di codice si nota come si definisce inizialmente un numero di *markers* da riconoscere sul volante (N_POINTS), il centro della *ROI* tramite un punto di coordinate x, y, la variabile *isRecording* che identifica in quale fase di esecuzione si trova l'algoritmo, le due liste di *Tracker* (che vedremo in seguito) che identificano l'insieme di punti rilevati in due frame successivi, una variabile di tipo *MatOfPoint* che rappresenta le coordinate spaziali dei punti che formano il contorno riconosciuto sul frame, una variabile *SteeringUtilities* che verrà utilizzata alla fine per il calcolo del *delta* di rotazione e una seconda lista di *Tracker* che identifica i punti *iniziali* riconosciuti.

```
public static void setBorders(int measuredHeight, int measuredWidth) {
        yBorder = measuredHeight;
        xBorder = measuredWidth;
        xRatio = xJcvBorder / xBorder;
        yRatio = yJvcBorder / yBorder;
    }
```

Tale metodo è utile per il passaggio di coordinate dal sistema di riferimento *OpenCV* a quello proprio di *Android*(5.2.2) come vedremo infatti l'activity che ingloba l'esecuzione dell'algoritmo deve farsi carico di trasformare tutti i calcoli effettuati nel sistema

di riferimento rispetto alla controparte. Richiamando questo metodo si riesce quindi a ridimensionare gli oggetti rispetto al sistema di riferimento Android.

```
public synchronized static void setROI(ImageView mDrawRect) {
        RectF r = getImageBounds(mDrawRect);
        int x = normalizeX(r.left);
        int y = normalizeY(r.top);
        int x1 = normalizeX(r.right);
        int y1 = normalizeY(r.bottom);
        objRect = new Rect(x, y, (x1 - x)/2, (y1 - y));

        mRoiCenter = new Point(x + (x1 - x)/4,
                               objRect.br().y - objRect.height/2);

    }

    public static RectF getImageBounds(ImageView imageView) {
        if(RoiView == null) RoiView = imageView;
        RectF bounds = new RectF();
        Drawable drawable = imageView.getDrawable();
        if (drawable != null) {
            imageView.getImageMatrix()
            .mapRect(bounds, new RectF(drawable.getBounds()));
        }
        return bounds;
    }
```

Tale metodo permette di modificare la posizione e la dimensione della *ROI* ogni qualvolta questa viene modificata dall'utente (tramite le gesture di *drag & drop* per la posizione e *pinch to zoom* per la dimensione). La *ROI* è infatti ottenuta tramite il costrutto primitivo di *Rect*, proprio di *OpenCV*, e disegnata come un cerchio inscritto in un rettangolo di raggio uguale a metà altezza del rettangolo. Questo risulta essere l'unico modo possibile poiché l'immagine, *mDrawRect*, manipolata dall'utente al fine di modificare l'area di acquisizione è rappresentata tramite un rettangolo sullo schermo. I metodi di *normalize* permettono la trasformazioni da coordinate Android a quelle di OpenCV.

```
private synchronized static void saveImageInformation() {
        Core.extractChannel(mYCbCr, channel, 0);
        meanBrightness  = Core.mean(channel);
        mBrightValues.addValue(meanBrightness.val[0]);
    }


    private static void adjustCameraParameters() {
        if(mBrightValues == null || mBrightValues.getN() == 0) return;
        double avg = mBrightValues.getMean();
        double delta = avg - meanBrightness.val[0];

        mBrightness += delta < 0 ? -1 : 1;
```

Questi due metodi sono invece utilizzati nella fase di salvataggio delle informazioni dell'immagine e di modifica dei parametri: il primo metodo infatti permette, tramite un costrutto

di *OpenCV* di estrapolare dal frame i valori di luminosità ed andarli ad includere in un costrutto apposito per calcolarne in seguito la media. Il secondo invece permette la correzione dell'immagine tramite la modifica del parametro *luminosità*: viene infatti estrapolata la media, calcolato il delta tra la luminosità corrente e quella appena calcolata ed andare a modificare il valore di una unità (a seconda del segno positivo o negativo del delta).

```
public interface ObjectTrackerListener {
        void onPointsReady();
        void onNotFound();
        void onROIChanged(Rect objRect);
    }
```

L'interfaccia *ObjectTrackerListener* è una interfaccia pubblica che permette alla classe rappresentante l'interfaccia camera di ricevere informazioni dalla classe adibita al tracking dei marker: i tre metodi infatti vengono invocati quando sono presenti i punti relativi ai marker da rappresentare, quando invece non si è riuscito ad identificare tali punti e quando la *ROI* viene modificata dall'utente.

```
public static class Tracker {
        private Point point;
        private Map<Point, Double> dist;
        private Scalar color;
        private boolean wasLost;
        private double [] delta;
        ...
}
```

La classe *Tracker* permette di memorizzare tutte le informazioni utili per quanto riguarda i marker da riconoscere: il marker in sé infatti viene rappresentato come un punto (*point*), contiene poi una *Mappa* che associa ad ognuno degli altri marker una distanza dal marker selezionato e un colore specifico associato a quel marker. Gli ultimi due parametri invece rappresentano la possibilità che il marker associato sia andato perso durante l'identificazione (caso già discusso prima) e il *delta* associato allo spostamento del marker rispetto al frame precedente. La classe ovviamente contiene tutti i metodi di *getter* e *setter* per le variabili esaminate.

```
public synchronized static boolean changeRecordingStatus() {
        if(!isRecording) {
            if (currFramePoints.size() != N_POINTS) return false;

            double cx = 0, cy = 0;
            double min = Double.MAX_VALUE;
            for (Tracker entry :  currFramePoints) {
                for(Tracker yrtne : currFramePoints) {
                    if(entry.equals(yrtne)) continue;
                    int dist = (int)Math
                                    .hypot(
                                      entry.getPoint().x -
                                      yrtne.getPoint().x,
                                      entry.getPoint().y -
```

```
                                        yrtne.getPoint().y
                                       );
                if(dist < min)
                    min = dist;
                entry.addDistance(yrtne.getPoint(), dist);
                entry.setColor(getScalar(
                               MaterialColorPalette
                                .getRandomColor("400")
                               )
                );
            }
            minDistBetweenPoints = min/2;

            initPoints.clear();
            initPoints.addAll(currFramePoints);
            setAvgDistance(initPoints);

        }
        isRecording = !isRecording;
        listener.onROIChanged(objRect);
        theta = 0;
        return true;
    }
```

Tale metodo permette, sfruttando le informazioni all'interno di ogni *Tracker* (*marker*) di memorizzare le informazioni relative alla posizione dei *primi* marker trovati: il metodo viene infatti invocato quando l'utente è sicuro che all'interno della *ROI* siano presenti tutti e quattro i marker da seguire e, per ognuno, il metodo calcola la distanza rispetto a tutti gli altri. Al termine del calcolo viene anche salvato il valore della distanza media tra i marker ed inseriti tutti in una lista apposita che rappresenta le posizioni iniziali (*initPoints*).

```
public synchronized static double elaborate(Mat mRgba,
                                            boolean inBackground) {
```

Il metodo **elaborate** rappresenta il contenitore principale dell'algoritmo di calcolo e per tale motivo verrà analizzato *step* dopo *step* per comprenderne meglio il funzionamento. Innanzitutto è bene notare che come secondo parametro, oltre alla matrice contenente le informazioni del frame, prenda un *booleano* denominato *inBackground*: tale valore sarà posto *true* nel momento in cui il metodo verrà utilizzato dal *service* che gestisce la fotocamera.

```
        mRgba.copyTo(mMask);
        double res = 0;
        mMask.setTo(new Scalar(0));
        int r = Math.min(objRect.width/2, objRect.height/2);
        Imgproc.circle(mMask, mRoiCenter, r, new Scalar(255,255,255), -1);
        try {
            if (mMask != null) {
                Core.bitwise_and(mRgba, mMask, mSelection);
            } else return 0;
        } catch(Exception e) {
```

```
            e.printStackTrace();
        }
        mContours.clear();
        Imgproc.cvtColor(mSelection, mYCbCr, Imgproc.COLOR_BGR2YUV);
```

In questa prima parte viene prelevata la matrice passata come parametro e salvata localmente per essere elaborata; viene allora definita la ROI, creando una *maschera* binaria sull'immagine ed eseguendo un *and* bit a bit con l'intera matrice: in questo modo l'unica parte visibile sarà quella all'interno della ROI. La maschera viene creata semplicemente tracciando un cerchio bianco sull'immagine stessa totalmente nera appena creata ($mMask$). Infine l'immagine viene convertita in YUV per essere successivamente elaborata.

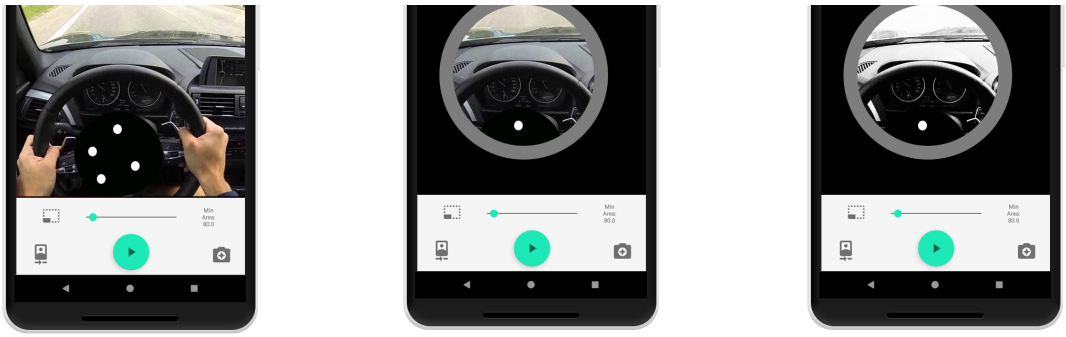


Figura 6.19. L'immagine iniziale viene "croppata" utilizzando il riferimento della ROI e successivamente convertita.

```
if(isRecording || inBackground) {
            int counter = 0;
            while(!findPoints(mSelection) && counter < 60) {
                adjustCameraParameters();
                counter++;
            }
        }else {
            if(!findPoints(mSelection)) {
                adjustCameraParameters();
                return 0;
            }
        }
```

Il prossimo passaggio rappresenta invece l'identificazione dei punti: come si evince dal codice ciò è fatto in due modi differenti a seconda che l'algoritmo sia in modalità *calcolo* o meno. Nel primo caso il metodo cicla fintanto che non trova i punti ed ad ogni iterazione viene modificato il parametro associato alla luminosità tramite la chiamata $adjustCameraParameters()$; ciò viene fatto per i 60 frame successivi. Nel secondo caso invece si tenta una sola volta poiché il controllo del parametro è comunque a portata di mano dell'utente tramite l'interfaccia grafica.

```
private static boolean findPoints(Mat mSelection) {

        mSelection.convertTo(mSelection, -1, mContrast, mBrightness);
        Imgproc.cvtColor(mSelection, mGray, Imgproc.COLOR_BGRA2GRAY);
        Imgproc.GaussianBlur(mGray, mGray, new Size(15, 15), 0);
        Imgproc.threshold(mGray, mGray, minThr, 255, Imgproc.THRESH_BINARY);
```

Il metodo inizialmente esegue delle trasformazioni dell'immagine per riuscire a minimizzare al minimo gli errori dovuti ai disturbi: viene innanzitutto settata la luminosità ideale, l'immagine viene convertita in scala di grigi per poi poter essere sfocata e *posterizzata*: in questo ultimo passaggio è quindi possibile rendere l'immagine *binaria*, quindi tutto ciò che sarà oltre una certa soglia, scelta dall'utente, verrà rappresentato tramite il bianco, il resto nero.

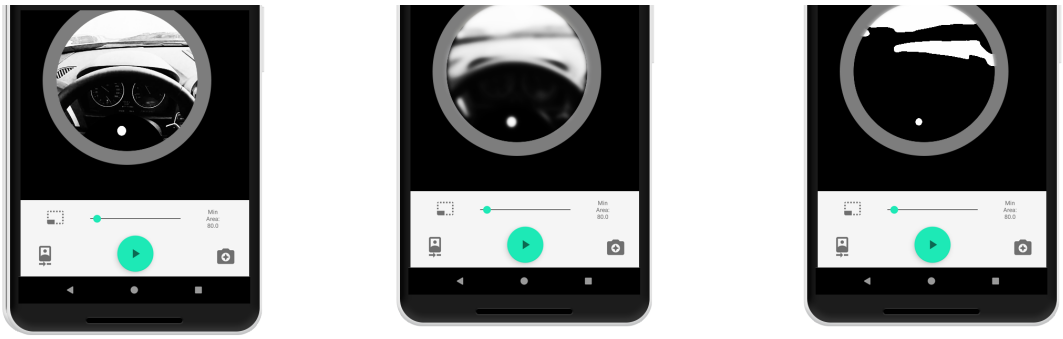


Figura 6.20. Il frame subisce delle trasformazioni per ridurre al minimo il rumore ed infine viene applicata la funzione *soglia*.

```
        Imgproc.findContours(mGray, mContours, mHierarchy,
                             Imgproc.RETR_EXTERNAL,
                             Imgproc.CHAIN_APPROX_SIMPLE);

        currFramePoints.clear();
        Iterator<MatOfPoint> each = mContours.iterator();

        while(each.hasNext()) {
            mContour = each.next();
            mMoments = Imgproc.moments(mContour);
            double x = mMoments.get_m10()/mMoments.get_m00();
            double y = mMoments.get_m01()/mMoments.get_m00();
            double area = mMoments.get_m00();

            if(Double.isNaN(x) || Double.isNaN(y))
                continue;
            else if(area > minArea) {
                mPoint = new Point(x, y);
                Tracker t = new Tracker(mPoint);
                currFramePoints.add(t);
```

```
            }
        }

        return currFramePoints.size() >= N_POINTS;
    }

    ...
    if(currFramePoints.size() >= N_POINTS -1) {
            listener.onPointsReady();
            saveImageInformation();
```

A questo punto viene invocata la funzione *findContours* che, preso in input il frame filtrato, una lista di *MatOfPoint* per fare lo *store* dei valori trovati, *mHierarchy* che rappresenta la gerarchia da seguire per ordinare i contorni in lista(default) e due parametri utili a suggerire come individuare i contorni, memorizza in *mContours* tutti i contorni trovati come matrici di punti: è possibile quindi iterare sulla lista e per ogni contorno calcolare, utilizzando la logica dei *Moments*, il baricentro del contorno e l'area. Facendo dei controlli su tali valori per essere sicuri di non avere niente di errato, viene allora aggiunto il contorno ad una lista di *Tracker*. Se vengono trovati almeno tre contorni (generici) viene inviato un segnale di avviso all'activity in esecuzione: in tal modo possono essere identificati sul display i baricentri dei contorni trovati. Infine viene salvata la luminosità corrente dell'immagine.

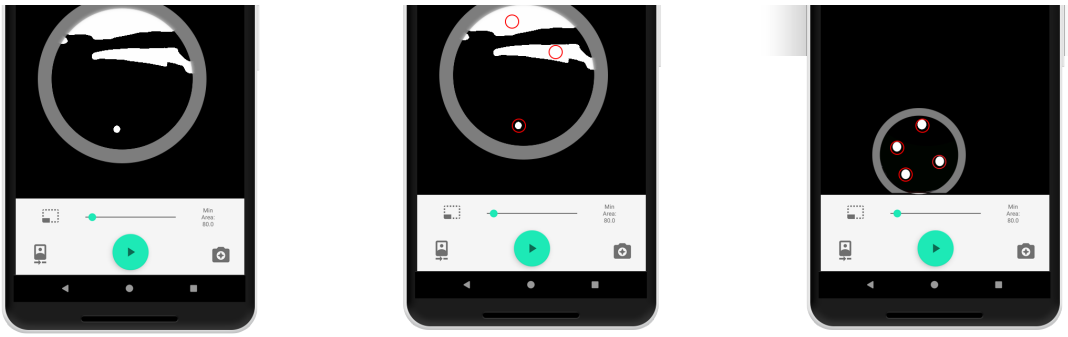


Figura 6.21. I contorni vengono mostrati all'utente come *punti* cerchiati: muovendo la *ROI* è possibile identificare quelli relativi ai marker sullo sterzo.

```
if(isRecording || inBackground) {
            for (Tracker tracker : currFramePoints) {
                double mindist = minDistBetweenPoints;
                int index = -1;
                for (Tracker t : prevFramePoints) {
                    if (t == null || t.getPoint() == null) continue;
                    double dist = Math.abs(
                                    Math.hypot(
                                     tracker.getPoint().x -
                                     t.getPoint().x,
```

```
                                tracker.getPoint().y -
                                t.getPoint().y
                                )
                            );
                    if (dist <= mindist) {
                        index = prevFramePoints.indexOf(t);
                        mindist = dist;
                        tracker.setDelta(
                                 new double[]{
                                  tracker.getPoint().x -
                                  t.getPoint().x,
                                  tracker.getPoint().y -
                                  t.getPoint().y});
                                 }
                        }
                if(index != -1) {
                    tracker.setColor(initPoints.get(index).getColor());
                    temp[index] = tracker;
                }
            }
```

Premendo quindi il pulsante *Play* comincia la fase di *calcolo* dell'algoritmo: viene allora eseguito un calcolo delle distanze minimi fra i marker dei due frame successivi e vengono assegnati come visto poco sopra: il vettore *temp* identifica invece quei marker *persi* o comunque non chiari in prima battuta che verranno ricostruiti in seguito.

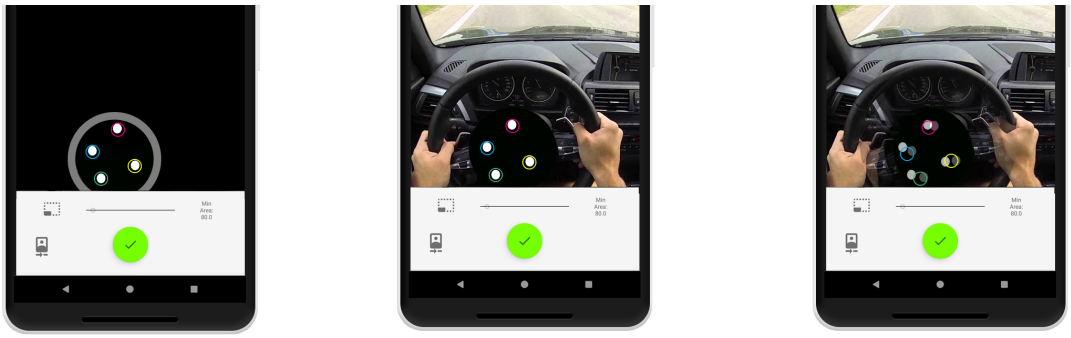


Figura 6.22. Le ultime due figure rappresentano l'interfaccia utente: nel momento in cui lo sterzo verrà ruotato i marker verranno *seguiti* dall'algoritmo e quindi cerchiati come identificati.

Successivamente se i punti verranno riconosciuti frame per frame l'algoritmo andrà avanti come illustrato nella schema iniziale; se invece vi è qualche problema dovuto al non riconoscimento inizia la fase di *recovery* dei punti persi:

```
for(int i = 0; i < lastGoodPoints.length; i++) {
                    if(!check[i]) {
                        double[] delta = null;
```

```
                for(int k = 0; k < check.length; k++) {
                    if(check[k]) {
                        delta = temp[k].getDelta();
                        if(delta != null)
                            break;
                    }
                }

                if(delta != null) {
                    if(((int)lastGoodPoints[i].getDelta()[0]
                     ^ (int)delta[0]) >= 0) /**Se i segni sono diversi **/
                        delta[0] = -delta[0];
                    if(((int)lastGoodPoints[i].getDelta()[1]
                     ^ (int)delta[1]) >= 0) /**Se i segni sono diversi **/
                        delta[1] = -delta[1];
                    Point p = new Point(lastGoodPoints[i].point.x + delta[0],
                     lastGoodPoints[i].point.y + delta[1]);
                    temp[i] = new Tracker(p);
                    temp[i].setDelta(delta);
                    temp[i].setColor(initPoints.get(i).getColor());
                    temp[i].setWasLost(true);

                    lastGoodPoints[i] = temp[i];
                }

            }

        }
```

La lista *lastGoodPoints* rappresenta gli ultimi punti individuati per *intero*: quindi con punti ricostruiti o comunque non persi. Iterando sulla lista vado a controllare, tramite un vettore parallelo (*check*) se alla stessa posizione corrisponde un punto perso nel frame corrente: in tal caso recupero le coordinate di x, y del frame precedente dello stesso punto e a queste vengono sommate i delta relativi agli altri punti. Infine viene quindi creato un nuovo *Tracker*, partendo dal punto calcolato, e salvato all'interno della lista di punti appena utilizzata.

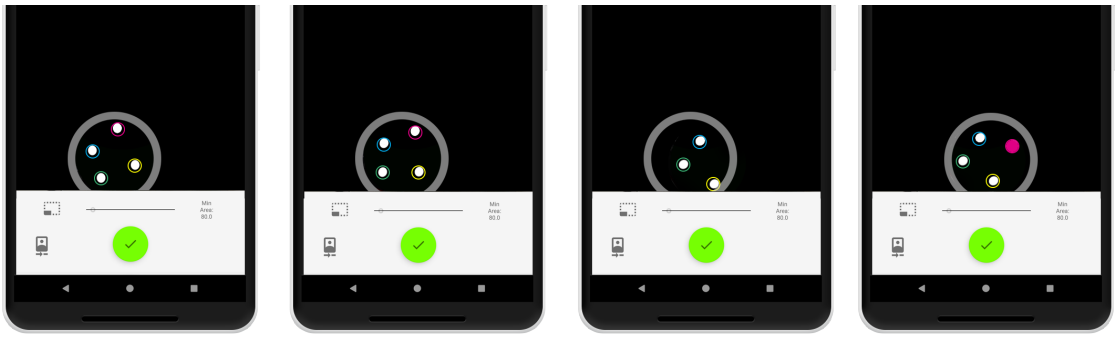


Figura 6.23. In questa immagine viene analizzato il caso di perdita di informazioni dovuto ad un temporaneo movimento: nella terza immagine viene infatti perso un marker e recuperato nella quarta. In quest'ultima si nota come anche la *ROI* viene ad essere traslata in base alle nuove coordinate dei quattro punti.

```
st = new SteeringUtilities(last, curr);
                        double var = st.calcSteeringAngle(mRgba);
```

Come ultima parte dell'algoritmo viene calcolato l'angolo tra i due frame rispetto ai punti contenuti e ritornato. Si andrà quindi ad analizzare adesso la classe *SteeringUtilities*:

```
public class SteeringUtilities {
    private double[][] coord1, coord2;
    private double[] centro1, centro2;
    private int N_PUNTI, N_COORD;


    public SteeringUtilities(Point[] prevFrame, Point[] currFrame){
        prevPoints = prevFrame.clone();
        coord1 = new double[2][prevFrame.length];
        for(int j = 0; j < prevFrame.length; j++) {
            coord1[0][j] = prevFrame[j].x;
            coord1[1][j] = prevFrame[j].y;
                }
            currPoints = currFrame.clone();
             coord2 = new double[2][currFrame.length];
        for(int j = 0; j < currFrame.length; j++) {
            coord2[0][j] = currFrame[j].x;
            coord2[1][j] = currFrame[j].y;
        }
        N_PUNTI = coord1[0].length;
        N_COORD = coord1.length;

        centro1 = new double[N_COORD];
        centro2 = new double[N_COORD];
```

La classe contiene al suo interno due matrici rappresentanti i valori dei quattro marker individuati, rispettivamente per il frame corrente e precedente. Stessa cosa per i due

centri. Il costruttore riceve come parametro i due array di *Point* dalla chiamata precedente e questi vengono memorizzati nelle due matrici; vengono inoltre inizializzate le strutture per contenere il centro dei due gruppi di punti.

```
public double calcSteeringAngle(Mat mRgba){

        // calcolo il centro di ciascun gruppo di punti

        for(int i = 0; i< N_COORD; i++){
            for(int j=0; j< N_PUNTI; j++){
                centro1[i] += coord1[i][j];
                centro2[i] += coord2[i][j];
            }
        }

        for(int i=0; i<N_COORD; i++){
            centro1[i] = centro1[i] / N_PUNTI;
            centro2[i] = centro2[i] / N_PUNTI;
        }

        // riporto i punti all'origine
        for(int i=0; i<N_COORD; i++) {
            for (int j = 0; j < N_PUNTI; j++) {
                coord1[i][j] = coord1[i][j] - centro1[i];
                coord2[i][j] = coord2[i][j] - centro2[i];
            }
        }

            Imgproc.circle(mRgba, new Point(
            centro2[0], centro2[1]),
            5, new Scalar(255,255,255), -1);
            RealMatrix diff1 = MatrixUtils.createRealMatrix(coord1);
            RealMatrix diff2 = MatrixUtils.createRealMatrix(coord2)
            .transpose();
            RealMatrix M = diff1.multiply(diff2);
            SingularValueDecomposition svd = new SingularValueDecomposition(M);
            RealMatrix R = svd.getU().multiply(svd.getVT());


            double teta = Math.atan2(
                           R.getEntry(1,0),
                           R.getEntry(0,0)) * (180.0 / Math.PI)
                        ;
            return teta;
```

Il metodo *calcSteeringAngle*, come visto in precedenza (5.2.5), riporta i punti all'origine, calcola le matrici necessarie ed estrapola da queste l'angolo di rotazione θ. Tale valore viene poi ritornato dal metodo.

6.4 Analisi delle classi

In questa sezione andremo ad analizzare le classi che compongono l'applicazione; la suddivisione interna delle classi è stata fatta seguendo la logica di utilizzo delle stesse: troveremo essenzialmente cinque tipi di classi:

- **Base Class:** questo tipo rappresenta le classi che vengono utilizzate per *modellare* i dati; sono per lo più classi *contenitore*, che implementano quindi strutture dati utili per raggruppare più dati all'interno di una sola classe, o anche utili ai fini di memorizzare dati all'interno del database interno dell'applicazione. Oltre a queste vi sarà anche una classe modellata appositamente per eseguire l'acquisizione in background del video, che quindi si interfaccia direttamente con l'hardware della fotocamera.
- **Helper Class:** come dice il nome, queste classi sono *d'aiuto* per lo sviluppatore in quei casi in cui è necessario svolgere delle *routine* predefinite per accedere a particolari parti dell'applicativo, come ad esempio le animazioni o l'accesso al database(nel nostro caso)(5.1.4).
- **Service Class:** contiene la classe che implementa il servizio per poter utilizzare la camera in background, interfacciandosi direttamente all'hardware.
- **Utility Class:** contiene le classi utili all'interfacciamento con il database e una classe per poter gestire la lettura e scrittura di determinati file interni.
- **Activity class:** come spiegato precedentemente (5.1.1) rappresentano la vera interfaccia con l'utente utilizzatore dell'applicazione; ogni classe activity corrisponde ad una "schermata" o comunque ad un "pezzo" di una schermata dell'app.

6.4.1 Classi Base

Come accennato precedentemente le classi base modellano le strutture dati utili per memorizzare dei dati statici o per interfacciare le classi principali dell'applicazione; in sostanza abbiamo tre tipi di classi base:

Entries Classes:

Tali classi rappresentano una struttura dati utile a memorizzare i valori acquisiti dai sensori del telefono per poter essere in seguito elaborati dall'algoritmo di stima del β, come si evince dal codice le cinque classi non fanno altro che memorizzare il valore per poi poterlo estrapolare in seguito, e di solito vengono organizzate in *Liste* di valori:

- **AccelEntry Class:**

```
public class AccelEntry implements Parcelable {

    private double x;
    private double y;
    private double z;

    ...
    }
```

Tale classe permette di memorizzare i valori delle componenti x, y, z delle accelerazioni proveniente dall'accelerometro. I metodi aggiuntivi presenti nella classe permettono di poter implementare una interfaccia presente in Android, *Parcelable* [2]

- **GpsEntry Class:**

```
public class GpsEntry implements Parcelable {

    private double speed;
    private double lat;
    private double lon;

    ...
    }
```

Anche questa classe permette di memorizzare i valori restituiti dal sensore GPS, in particolare velocità, latitudine e longitudine. Allo stesso modo implementa *Parcelable*[1] per poter essere trasmessa via *intent*.

- **GyroEntry Class:**

```
public class GyroEntry implements Parcelable {

    private double x;
    private double y;
    private double z;

    ...
        }
```

Permette di memorizzare le componenti x, y, z del sensore giroscopio.

[2]Implementare una interfaccia di questo tipo è utile in quei casi in cui voglio scambiare dei dati *non* primitivi, quindi una classe, tra diverse Activity o Service. L'oggetto *Parcel* infatti è un contenitore che contiene al suo interno il dato e permette di poter essere $flattened$, quindi una sorta di serializzazione, da una parte per poi essere $unflattened$ dalla parte opposta che necessita il dato. Di solito quando si vuole scambiare una struttura tramite gli *intent*, la classe che deve essere passata deve necessariamente implementare questa interfaccia.

- **StEntry Class:**

```
public class StEntry implements Parcelable {
    private double theta;

        ...
        }
```

Permette di memorizzare il valore θ dell'angolo volante.

CarParameters Class:

La classe è utilizzata per memorizzare e modificare i valori *statici* delle variabili associate al segmento di veicolo scelto per la prova:

```
public class CarParameters {
    double  C1 , //[N/rad] rigidezza deriva pneumatici anteriori
            C2,     //[N/rad] rigidezza deriva pneumatici posteriori
            a1,      //[m] semipasso ant
            a2,      //[m] semipasso post
            tau,    //rapporto sterzo anteriore
            J,   //[kg*m^2] mom di inerzia cassa
            m;      //[kG] massa veicolo
    String param;   //univoco per ogni segmento di veicolo
    int img;        //Drawable associato al segmento
    ...
    }
```

HardwareCamera Class:

```
public class HardwareCamera implements ICamera,
        Camera.PreviewCallback {

    private final Context context;
    private final CameraService user;
    private Camera mCamera;
    private int mFrameWidth;
    private int mFrameHeight;
    private CameraAccessFrame mCameraFrame;
    private CameraHandlerThread mThread = null;
    private SurfaceTexture texture = new SurfaceTexture(0);

    // needed to avoid OpenCV error:
    // "queueBuffer: BufferQueue has been abandoned!"
    private byte[] mBuffer;

    ...
```

Rappresenta la classe utilizzata dal Servizio per mantenere attiva la camera in background. Implementa i metodi di due interfacce: *ICamera*, rappresenta una interfaccia creata appositamente per implementare due metodi generici:

```
interface ICamera {

    void connect();

    void release();

}
```

- Invocando **connect()** viene inizializzato l'oggetto *mThread* della classe, facendo quindi partire un thread che si occupa della fase di inizializzazione della camera (oggetto *mCamera*): vengono raccolti tutti i dati necessari a capire il tipo di camera che si sta utilizzando e come rappresentare l'anteprima dell'immagine raccolta dalla camera: dimensione dello schermo, risoluzione del *frame* da acquisire, se si sta utilizzando la camera anteriore o posteriore. Durante questa fase viene anche inizializzato l'oggetto *mCameraFrame*, molto importante poiché si occupa di trasferire ed interpretare i dati *raw* ricevuti dalla camera come *byte*[] ed elaborarli come matrici per essere interpretati da *OpenCV*. Infine viene inizializzato anche l'oggetto *texture*, che nel nostro caso è praticamente inutile, poiché rappresenta la *preview* dell'immagine che viene mostrata sullo schermo del dispositivo, ma è comunque necessario crearla poiché l'interfacciamento base della camera consta di callback riferite a questo oggetto.
- **release()** invece è il metodo utilizzato per fare "clean-up" degli oggetti una volta che il servizio smette di funzionare; è infatti invocato all'interno della callback *onDestroy*() del service stesso. Non fa altro che fermare l'esecuzione del thread (*mThread*), forzandolo a smettere e distruggendolo; in più forza anche la terminazione di *mCamera*, smettendo così di acquisire dati da mostrare sullo schermo e distruggendo anche questo oggetto.

La seconda interfaccia che viene implementata nella classe è *Camera.PreviewCallback*, interfaccia (ormai deprecata) della classe Android che permette appunto di mostrare le acquisizioni della camera sul display: una domanda sorge a questo punto spontanea, essendo un servizio in background, quindi non necessitando di mostrare nulla all'utente, perché è stato implementato in questo modo? La risposta è che *OpenCV* non lavora sui dati in byte trasferiti dal sensore della camera al sistema operativo, bensì lavora sulla *Preview* dell'immagine che viene mostrata poiché rappresenta la fase di passaggio da byte di dati ad oggetto vero e proprio; la scelta di implementazione è stata quindi "forzata" ad utilizzare le callback classiche di un'interfacciamento con la camera del dispositivo. Il metodo più importante dunque di questa seconda interfaccia è:

```
void onPreviewFrame(byte[] data, Camera camera);
```

in cui l'array di byte *data* viene passato al metodo *put*(..) dell'oggetto *CameraAccessFrame*.

```
public class CameraAccessFrame implements CameraFrame {
        private Mat mYuvFrameData;
        private Mat mRgba;
        private int mWidth;
        private int mHeight;
        private Bitmap mCachedBitmap;
        private boolean mRgbaConverted;
        private boolean mBitmapConverted;

                ...
                }
```

Di fondamentale importanza è anche l'oggetto *mCameraFrame* dichiarato come sopra, che implementa l'interfaccia *CameraFrame* costruita appositamente *estendendo* il funzionamento della classe base di *OpenCV*: **CameraBridgeViewBase** [3]

```
public interface CameraFrame extends CameraBridgeViewBase.CvCameraViewFrame {
        @Override
    Bitmap toBitmap();

    @Override
    Mat rgba();

    @Override
    Mat gray();
}
```

I tre metodi vengono dunque sovrascritti per permettere una facile conversione tra oggetti di tipo *byte Array*, ricevuti in ingresso, ed oggetti processabili da *OpenCV*. Invocando il metodo *put* appartenente all'oggetto tipo *Mat* viene creata una matrice partendo da un array di byte, ricevuti dalla camera, il tutto in modo trasparente. L'oggetto sarà dunque processabile direttamente dall'algoritmo implementato in *OpenCV*.

```
public synchronized void put(byte[] frame) {
            mYuvFrameData.put(0, 0, frame);
            invalidate();
        }
```

L'implementazione dell'intera classe *HardwareCamera* verrà vista successivamente quando verrà trattato il *Service* apposito.

[3]This is a basic class, implementing the interaction with Camera and OpenCV library. The main responsibility of it - is to control when camera can be enabled, process the frame, call external listener to make any adjustments to the frame and then draw the resulting frame to the screen. (*OpenCV Documentation*)

6.4.2 Classi Helper

AnimationHelper Class:

Come si evince dal nome la classe, contenente esclusivamente metodi statici, è utile per gestire le animazioni all'interno dell'applicazione richiamando i metodi appositamente costruiti. Particolarmente interessanti sono i metodi di animazione che riguardano i *Drawable* [4]

```
public static void animateDrawable(View v) {
        Drawable drawable = v.getDrawable();
        Animatable anim;
        if (drawable instanceof Animatable) {
            ((Animatable) drawable).start();
        }
    }
```

Il metodo estrae il *Drawable* dalla *View*, e se questo corrisponde ad uno speciale tipo di drawable (*AnimatedVectorDrawable* che contiene al suo interno un oggetto *Animator*, che estende il concetto generico di *Animatable*) questo ne inizia l'animazione.

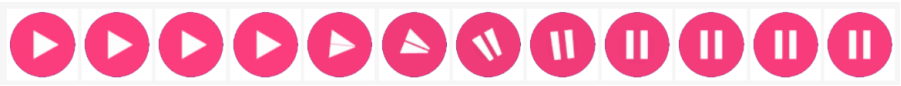


Figura 6.24. Esempio di animazione di drawable.

[4] A Drawable is a general abstraction for "something that can be drawn." Most often you will deal with Drawable as the type of resource retrieved for drawing things to the screen; the Drawable class provides a generic API for dealing with an underlying visual resource that may take a variety of forms. Unlike a View, a Drawable does not have any facility to receive events or otherwise interact with the user. (*Android Documentation*)

```
public static void circularRevealForBottomView(...) {

        ...

         Animator animator;
         if(reverse)
             animator= ViewAnimationUtils.createCircularReveal(viewToReveal,
              cx, cy, finalRadius, 0);
         else
             animator= ViewAnimationUtils.createCircularReveal(viewToReveal,
              cx, cy, 0, finalRadius);
         animator.setInterpolator(new FastOutSlowInInterpolator());
         animator.setDuration(500);
         animator.start();

         ...
                 }
```

Questa animazione è comunemente utilizzata per far apparire un layout o una generica *view* in background, in primo piano.

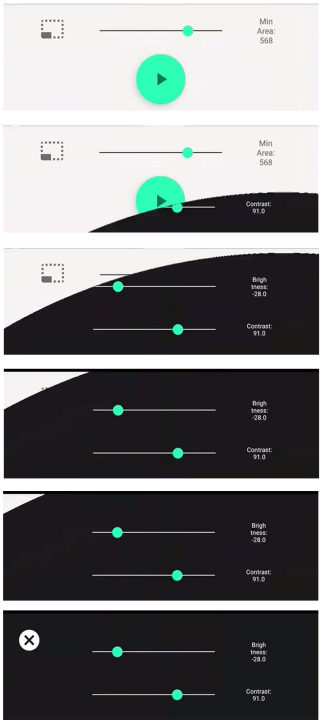


Figura 6.25. Esempio utilizzato nel caso di Car-App

```
public static void showTutorialFor(Activity activity, View target,
        String title, String description) {
        TapTargetView.showFor(activity,
                TapTarget.forView(target, title, description)
                        .outerCircleColor(R.color.secondaryColor)
                        .outerCircleAlpha(0.75f)
                        .targetCircleColor(R.color.cardview_light_background)
                        ...
                        }
```

Tramite l'utilizzo della libreria *TapTargetView* è stato implementato un tutorial di base per l'utilizzo dell'applicazione, per richiamare il tutorial su certi *target* dell'applicazione è stato definito un metodo preciso che assegna vari parametri tra i quali: didascalie, colori dei targets, durata, trasparenza...

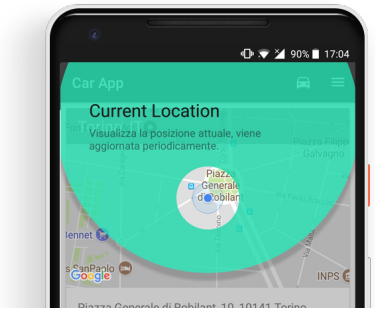


Figura 6.26. Tutorial implementato in Car-App

6.4.3 Classi Service

Come spiegato precedentemente (5.1.2) i *Services* vengono utilizzati per svolgere delle operazioni durature nel tempo e non hanno bisogno di interagire con l'utente; vengono quindi eseguiti in background. Nel caso particolare dell'applicazione sviluppata è stato utilizzato solamente un service (quello inerente al segnale GPS è stato implementato con una libreria specifica [19].

CameraService Class:

```
public class CameraService extends Service{
    private HardwareCamera mCamera;
    private LinkedBlockingQueue<Mat> mFrames;
    private Thread mThread;
    private int mCameraId;

    ...
```

L'implementazione segue quella standard di un qualunque service: viene effettuato l'*Override* dei metodi:

```
@Override
    public IBinder onBind(Intent intent);

@Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        Handler h = new Handler();
        h.postDelayed(new Runnable() {
            @Override
            public void run() {
                int camera = Camera.CameraInfo.CAMERA_FACING_BACK;
                if(PreferenceManager.getDefaultSharedPreferences(
                                        getApplicationContext())
                                        .getBoolean(
                                        SettingsActivity
                                        .KEY_PREF_CAMERA, false))
                    camera = Camera.CameraInfo.CAMERA_FACING_FRONT;
                mCameraId = camera;
                mCamera = new HardwareCamera(CameraService.this,
                 CameraService.this,
                 mCameraId);
                mCamera.connect();
                mThread = new Thread(new BackgroundCamera());
            }
        }, 500);
                return super.onStartCommand(intent, flags, startId);
```

In particolare viene fatto il check della libreria *OpenCV*, dopo viene lanciato un thread secondario con ritardo ($delay = 500ms$) per evitare errori di sovrapposizione dell'esecuzione. All'interno dell'*Handler* [5] viene inizializzato l'oggetto *Camera* (6.4.1) e specificato se bisogna utilizzare la camera frontale o quella posteriore (il check viene fatto leggendo un particolare valore di cui parleremo in seguito). Per utilizzare l'oggetto e quindi poter elaborare i dati della camera bisogna implementare le callback relative alle interfacce utilizzate nella classe *HardwareCamera*:

```
public void onPreviewFrame(HardwareCamera.CameraAccessFrame mCameraFrame) {
try {
            Mat m = mCameraFrame.rgba();
            if(mCameraId == 1)
                Core.flip(m, m, 1);
            mFrames.put(m);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
```

[5] A Handler allows you to send and process Message and Runnable objects associated with a thread's MessageQueue. Each Handler instance is associated with a single thread and that thread's message queue.There are two main uses for a Handler: (1) to schedule messages and runnables to be executed as some point in the future; and **(2)** to enqueue an action to be performed on a different thread than your own. (*Android Documentation*

```
        if(!isRunning) {
                mThread.start();
            }


    }
```

Ciò che viene eseguito dentro la callback richiama il pattern $Producer - Consumer$ viene catturato il frame corrente passato come parametro:

```
Mat m = mCameraFrame.rgba();
```

Viene controllato l'ID della camera per capire se bisogna ruotare o meno il frame (il motivo verrà illustrato successivamente) e immediatamente viene inserito in un particolare costrutto:

```
mFrames.put(m);
```

L'oggetto in questione è istanziato come **LinkedBlockingQueue<Mat>**: rappresenta un costrutto $FIFO$ ideale nel caso in cui si vogliano condividere dei dati tra due thread diversi, che vengono identificati come

- **Producer Thread**: rappresenta nel nostro caso il thread appena visto, raccoglie il frame dalla callback e lo inserisce in coda.
- **Consumer Thread**: è invece il thread invocato al termine del metodo *onStardCommand* che preleverà il dato dalla coda e lo elaborerà. Tale metodo contiene all'interno un ciclo $infinito$ che smetterà di eseguire nel momento in cui il *Service* non verrà arrestato.

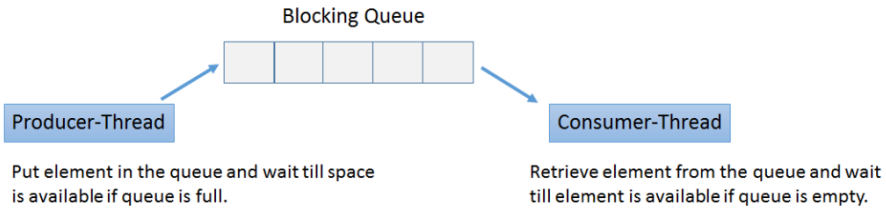


Figura 6.27. Costrutto *BlockingQueue*

Il tutto avviene in modo sincronizzato poiché all'interno contiene dei costrutti di sincronizzazione a basso livello basati su $wait - notify$ in modo da capire quando la coda contiene degli elementi e quindi quando è possibile prelevarne qualcuno. In tal caso il costrutto è dichiarato **Thread-Safe**. Il thread consumatore che si occuperà di svolgere i calcoli sui frame rimarrà quindi in attesa nel momento in cui il produttore non abbia più frame da passare; ciò evita molti dei problemi che nascono utilizzando un approccio $multi - thread$ nella stesura del codice.

Il metodo invocato dal consumatore è il seguente:

```
private class BackgroundCamera implements Runnable {

        @Override
        public void run() {
           ...
           while (true) {
              mFrame = mFrames.take();
              if(mFrame == null) continue;
              theta += ObjectTracker.elaborate(mFrame, true);
              final Intent i = new Intent("angle_update");
              i.putExtra("theta", -theta);
              sendBroadcast(i);
        }
}
```

Qui figura il comando:

```
mFrame = mFrames.take();
```

Effettua un *look − up* all'interno della coda condivisa, se è presente un valore lo estrae (quindi quest'ultimo viene *eliminato* dalla testa della coda) e lo ritorna come parametro, viene quindi incapsulato in una variabile e successivamente utilizzato come parametro da un metodo altrettanto fondamentale, in cui risiede tutto l'algoritmo di calcolo e tracciamento dei markers sul volante. Se invece non trova nessun valore questo rimarrà in attesa, una attesa chiamata senza *polling* ossia: il thread non continuerà ad effettuare dei controlli sulla coda a ogni ciclo ma in un certo senso si *addormenter*à su quella riga e verrà *svegliato* nel momento in cui un dato arriva sulla coda. Tale meccanismo permette di mantenere un alto grado di performance, poiché non vengono consumati cicli inutili di CPU dal processore.

La chiamata al metodo *elaborate* sulla classe *ObejectTracker* verrà analizzata successivamente.

Dopo aver elaborato il frame ed aver calcolato il θ corrente, tale valore è inserito dentro un *Intent* (il segno − che figura davanti al θ è indice del fatto che come convenzione degli angoli è stata utilizzata quella opposta a ciò che utilizza l'algoritmo di calcolo) ed inviato in *broadcast* per essere quindi catturato da un *BroadcastReceiver*(5.1.3).

6.4.4 Classi Utilities

Nella categoria *utility* rientrano vari tipi di classi: alcune atte a semplificare operazioni ripetitive e che necessitano ogni volta di una fase di inizializzazione, quali ad esempio la gestione del *database*; altre implementano gli algoritmi principali sviluppati per l'applicazione, quali il calcolo del β (angolo di assetto) e del θ (angolo volante).

DatabaseUtilities Class:

Come si evince dal nome la classe è utilizzata per gestire tutte le operazioni di lettura/scrittura sul database interno (5.1.4).

```
public class DatabaseUtilities {
    public DatabaseUtilities(Context ctx) {
           this.ctx = ctx;
       }

    public DatabaseHelper newHelper(){
          DatabaseHelper db = new DatabaseHelper(ctx);
          return db;
      }

      ...
}
```

Una volta istanziato l'oggetto *DatabaseUtilities* è possibile richiamare il metodo **newHelper()** che restituisce:

```
public static class DatabaseHelper extends SQLiteOpenHelper {

        // If you change the database schema, you must increment the database version.
        public static final int DATABASE_VERSION = 8;
        public static final String DATABASE_NAME = "database.db";

        public DatabaseHelper(Context context) {
            super(context, context.getExternalFilesDir(null)  + "/"
            + DATABASE_NAME, null, DATABASE_VERSION);
        }

        ...
}
```

L'oggetto in questione è uno standard utilizzato dal sistema operativo per *aiutare* il programmatore con le operazioni di creazione e update del database (con update si intende un aggiornamento dello *scheletro* del db, nel momento in cui si volesse aggiungere una colonna o una tabella bisogna incrementare il numero corrispondente a *DATABASE VERSION* per far capire all'applicazione che deve ricostruire il database per intero una volta istanziato). Le operazioni base di questa classe che vengono implementate sono:

- **Creazione:**

```
public void onCreate(SQLiteDatabase db) {

            db.execSQL(GPSEntry.SQL_CREATE_ENTRIES);
            db.execSQL(AccelEntry.SQL_CREATE_ENTRIES);
            db.execSQL(GyroEntry.SQL_CREATE_ENTRIES);
            db.execSQL(MagnetEntry.SQL_CREATE_ENTRIES);
            db.execSQL(SteeringWheelEntry.SQL_CREATE_ENTRIES);
            db.execSQL(ResultEntry.SQL_CREATE_ENTRIES);
            db.execSQL(ResultEntry.SQL_CREATE_INDEX);

        }
```

Il parametro passato alla *query* corrisponde ad un semplice comando SQL di tipo *CREATE TABLE*. Vengono create tante tabelle quanti sono i dati da raccogliere poiché la prima implementazione dell'algoritmo prevedeva un calcolo in *post* dell'angolo di assetto β. Con l'ultima versione invece il calcolo viene fatto in realtime, quindi non vi è bisogno di memorizzare i dati su tabelle; le uniche utilizzate saranno *ResultEntry*, per memorizzare le prove effettuate e *SteeringWheelEntry* per memorizzare le posizioni iniziali dei markers. Da notare anche la riga:

```
db.execSQL(ResultEntry.SQL_CREATE_INDEX);
```

A cui corrisponde il comando:

```
CREATE INDEX result_idx ON risultati_prove (id_prova);
```

Creando un indice sulla colonna *id* della tabella, questo permette di velocizzare di un fattore *estremamente* alto il tempo di lettura del dato associato a quell'indice: durante la lettura dei dati relativi ad una singola prova, questi vengono selezionati in base all'id *univoco* di ogni prova; creando un indice sulla colonna non avrò problemi ad effettuare il *retrive* di dati estremamente corposi, come nel caso di prove lunghe ore (si parla di circa 200.000 -250.000 record, letti in appena qualche secondo).

Benchmark results

CPU time no index	CPU time with index	Elapsed time no index	Elapsed time with index
94	0	7006	60
15	0	6267	30
31	0	6000	61
47	0	6218	32
235	0	6276	31

Figura 6.28. Confronto relativo ad un Database di 100.000 records casuali.

- **Upgrade:**

```
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
            db.execSQL(GPSEntry.SQL_DELETE_ENTRIES);
            db.execSQL(AccelEntry.SQL_DELETE_ENTRIES);
            db.execSQL(GyroEntry.SQL_DELETE_ENTRIES);
            db.execSQL(MagnetEntry.SQL_DELETE_ENTRIES);
            db.execSQL(SteeringWheelEntry.SQL_DELETE_ENTRIES);
            db.execSQL(ResultEntry.SQL_DELETE_ENTRIES);
            onCreate(db);
        }
```

Con la callback *onUpgrade* viene sostanzialmente distrutto il database corrente e ricreato seguendo lo schema del nuovo db passato come parametro.

StorageUtility Class:

```
public class StorageUtility {

    private static final String FILE_ACC_X_NAME = "acc_x.txt";
    private static final String FILE_ACC_Y_NAME = "acc_y.txt";
    private static final String FILE_ACC_Z_NAME = "acc_z.txt";

    private static final String PATH = Environment.getExternalStorageDirectory()
    .getAbsolutePath() + "/Android/data/it.polito.dimeas.carapp/raw_data/";

...
}
```

La classe è utilizzata principalmente per la lettura/scrittura dei valori di configurazione iniziale durante la prova. I tre file corrispondono infatti ai tre valori di accelerazioni lette durante tale periodo, mentre il *path* indica dove dover salvare/leggere tali file. I metodi principali sono essenzialmente due:

- **Lettura:**

```
public static void readFileAccs(List<Point3D_F32> st, List<Point3D_F32> dyn) {

        try {
            InputStream i_x = new FileInputStream(
                                PATH.concat(FILE_ACC_X_NAME)
                            );
            InputStream i_y = new FileInputStream(
                                PATH.concat(FILE_ACC_Y_NAME)
                                );
            InputStream i_z = new FileInputStream(
                                PATH.concat(FILE_ACC_Z_NAME)
                                );

            BufferedReader reader_x = new BufferedReader(
                                        new InputStreamReader(i_x));
            String raw_x = reader_x.readLine();

            BufferedReader reader_y = new BufferedReader(
                                        new InputStreamReader(i_y));
            String raw_y = reader_y.readLine();

            BufferedReader reader_z = new BufferedReader(
                                        new InputStreamReader(i_z));
            String raw_z = reader_z.readLine();

            ...
            }
```

La lettura dei file avviene utilizzando i costrutti di *InputStream*, che rappresenta una classe *astratta* contenente tutti i metodi utili a leggere da uno stream di dati. Una volta aperti i tre stream, uno per ogni file, i dati vengono letti riga dopo riga e immagazzinati nelle due strutture passate come parametro: la prima rappresenta i punti letti durante la fase *statica* di inizializzazione, il secondo durante la fase di *accelerazione* (il tutto verrà spiegato meglio in seguito). Al termine i tre stream vengono chiusi.

- **Scrittura**

```
public static void writeFileAccs(List<Point3D_F32> st, List<Point3D_F32> dyn) {
        File folder = new File(PATH);
        if(!folder.exists()) {
            if(!folder.mkdirs())
                return;
        }

        File file_x = new File(PATH, FILE_ACC_X_NAME),
        file_y = new File(PATH, FILE_ACC_Y_NAME),
        file_z = new File(PATH, FILE_ACC_Z_NAME);
        if(!file_x.exists())
            try {
                if(!file_x.createNewFile())
                    return;
            } catch (IOException e) {
                e.printStackTrace();
            }
        if(!file_y.exists())
            try {
                if(!file_y.createNewFile())
                    return;
            } catch (IOException e) {
                e.printStackTrace();
            }
        if(!file_z.exists())
            try {
                if(!file_z.createNewFile())
                    return;
            } catch (IOException e) {
                e.printStackTrace();
            }

        try {
            OutputStream outputStream_x = new FileOutputStream(
            PATH.concat(FILE_ACC_X_NAME)
            );
            OutputStream outputStream_y = new FileOutputStream(
            PATH.concat(FILE_ACC_Y_NAME)
            );
            OutputStream outputStream_z = new FileOutputStream(
            PATH.concat(FILE_ACC_Z_NAME)
            );

            BufferedWriter writer_x = new BufferedWriter(
                                        new OutputStreamWriter(
                                              outputStream_x
                                            )
                                        );
            BufferedWriter writer_y = new BufferedWriter(
```

```
                                        new OutputStreamWriter(
                                              outputStream_y
                                            )
                                        );
                BufferedWriter writer_z = new BufferedWriter(
                                        new OutputStreamWriter(
                                              outputStream_z
                                            )
                                        );

    ...
    }
```

Così come in lettura, anche in scrittura vengono utilizzati gli stream di dati, questa volta in senso *opposto*: dalla struttura dati vado a scrivere su un file fisico. La differenza è all'inizio del metodo dove vengono effettuati numeri check per evitare errori grossolani, viene quindi creata la cartella, se questa non esiste; allo stesso modo vengono controllati i file che, nel momento in cui non sono ancora stati creati, vengono aperti e scritti direttamente.

Un ulteriore metodo è utilizzato invece per la scrittura del file *CSV* nel momento in cui l'utente decida che, al termine di ogni prova, voglia ricevere i risultati via mail.

```
public static void writeFileLog(final String path, String filename,
                                 final String[] columns, final String[] content,
                                 String id) {
        File file = new File(path);
        if(!file.exists()) {
            if(!file.mkdirs())
                return;
        }

        file = new File(path, filename);
        if(!file.exists())
            try {
                if(!file.createNewFile())
                    return;
            } catch (IOException e) {
                e.printStackTrace();
            }
        CsvWriter writer = new CsvWriter();
        writer.setTextDelimiter(',');
        try (CsvAppender csvAppender = writer.append(new FileWriter(
        path + filename))) {
            // header
            csvAppender.appendLine(columns);
            for (String data : content) {
                data = new StringBuilder(data).insert(0, id + ";").toString();
                String[] token = data.split(";");
                csvAppender.appendLine(token);
            }
```

```
            csvAppender.close();

            ...
}
```

La routine di passaggi iniziali è molto simile al caso di scrittura di un file normale, per convertire il tutto in *CSV* si utilizza una libreria specifica che si occupa di mantenere il formato corretto con pochissime righe di codice. Basta infatti specificare il *delimitatore* per le colonne ed effettuare l'*append* dei dati come array di stringhe.

6.5 Le classi Activity

Come spiegato in precedenza (5.1.1) le activity rappresentano, ognuna, una *schermata* dell'applicazione con cui l'utente può interagire; nel nostro caso andremo a commentare una per una le activity principali e il menù che, nonostante non sia implementato come activity vera e propria, è parte integrante della UI.

6.5.1 Main Activity

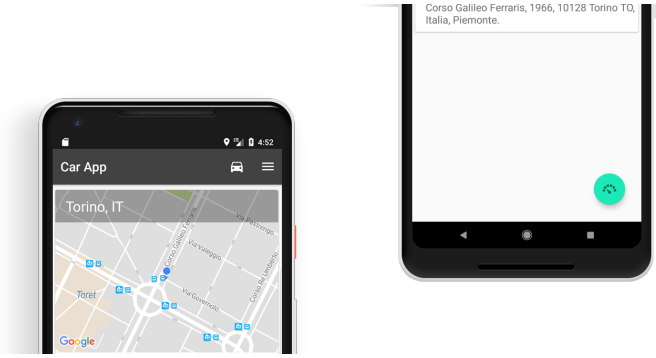


Figura 6.29. Screenshot Main Activity.

La *MainActivity* rappresenta la schermata principale dell'applicazione: da qui è possibile esplorare tutte le funzionalità e modificare le impostazioni; è anche possibile cambiare il *segmento* di veicolo, scegliendo dall'apposito menu, o andare a visualizzare la lista delle prove effettuate. La mappa in alto serve invece a mostrare all'utente la posizione corrente e, nel caso in cui il veicolo si muova, questo traccerà il percorso sulla mappa.

6.5.2 CameraTracker Activity

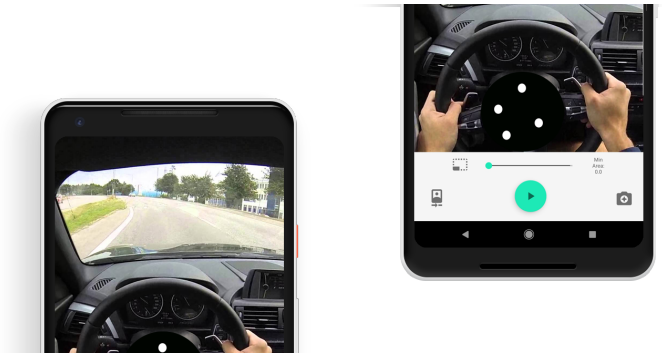


Figura 6.30. Screenshot CameraTracker Activity.

Come si evince dal nome e, come precedentemente spiegato, tale classe è utilizzata per effettuare la lettura ed il *setup* dell'angolo volante: tramite un'interfaccia molto semplificata è possibile posizionare la *ROI*, modificare i parametri della camera stessa, *switchare* in camera frontale, effettuare il *lock* delle posizioni dei marker per iniziare la prova.

6.5.3 Main Activity (Test in corso)

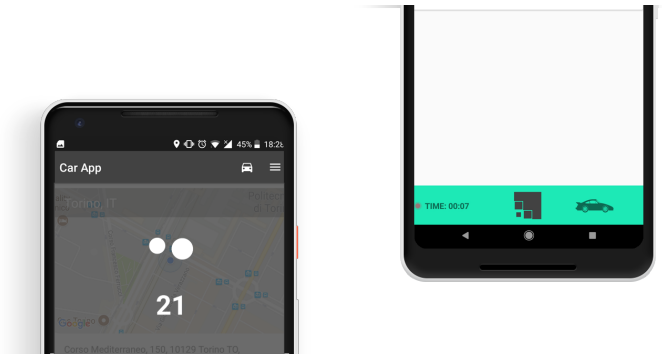


Figura 6.31. Screenshot Main Activity durante la prova.

La schermata rappresentata in alto è invece quella base durante l'inizio di una prova: vengono infatti mostrate a schermo, tramite dei messaggi e un timer, le istruzioni da seguire nel caso di inizializzazione della matrice di rotazione (6.3.2). Una volta terminata la procedura in basso partirà un *timer* con il conteggio dei minuti della prova e un pulsante al centro per stoppare la prova stessa. Una volta terminata verranno mostrati i risultati ottenuti.

6.5.4 Results Activity

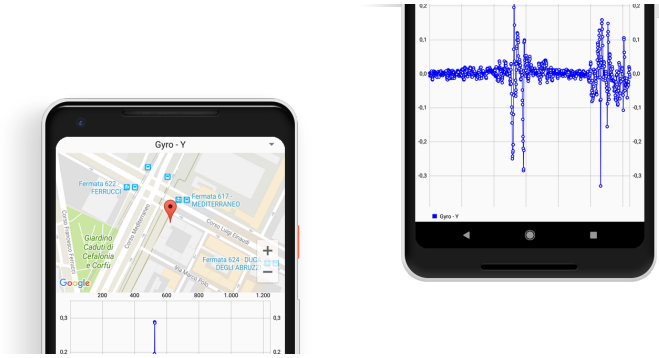


Figura 6.32. Screenshot Results Activity.

In questa *activity* sono mostrati i dati raccolti e *plottati* su un grafico, in basso; in alto è invece presente una mappa su cui è possibile selezionare dei segmenti di prova: il grafico sarà adattato infatti a contenere solamente i risultati inerenti al segmento selezionato. E' così possibile fare un'analisi più dettagliata della prova.

6.5.5 About Activity

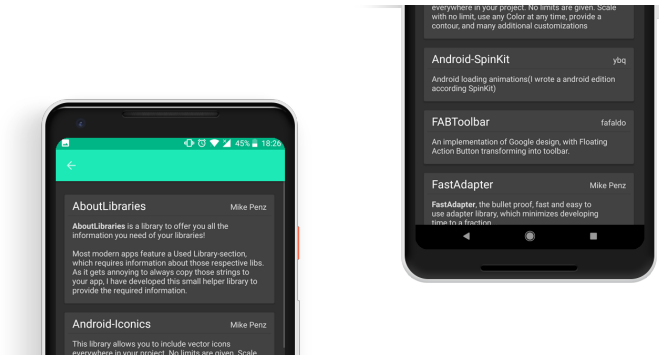


Figura 6.33. Screenshot About Activity.

Contiene tutte le *references* alle librerie utilizzate nell'applicazione: e' possibile aggiungere dinamicamente nuove *entry* semplicemente editando un file .XML all'interno del progetto. Se cliccata, ogni card *reindirizza* alla pagina dello sviluppatore della libreria.

Parte III

Terza Parte

Capitolo 7

Risultati Finali

In questa terza ed ultima parte si andranno ad analizzare i risultati ottenuti facendo riferimento a delle prove simulate su software *CarMaker*, in prima battuta, e successivamente andando anche ad analizzare delle prove reali direttamente con il terminale montato sul veicolo. Per quanto riguarda i risultati c'è da tener presente che nelle prove simulate:

- L'algoritmo funziona in maniera *generica*: deve essere in grado di effettuare misurazioni su qualunque tipo di manovra: le matrici di covarianza (P e Q) non possono essere settate a priori; così come il valore di R.
- Il veicolo utilizzato durante la simulazione non è implementato in maniera precisa nel segmento auto utilizzato durante il calcolo sullo Smartphone: vari parametri sono comunque simili ma altri non risultano minimamente comparabili.

Tali differenze possono portare a valori differenti ma comunque paragonabili in termini di andamento ed ordine di grandezza.

7.1 Prove Simulate

Figura 7.1. Esempio di modello di veicolo su CarMaker.

Le prove che seguono, come accennato poc'anzi, sono state dapprima effettuate sul simulatore; i risultati sono quindi molto precisi ed in un certo senso ideali: le misure prese in ingresso da *CarMaker* sono derivate da modelli di veicoli reali mentre sullo smartphone Android la precisione dell'elaborazione dati è limitata a segmenti di veicoli (6.4.1).

7.1.1 Prova 1: Curva semplice

La prima simulazione che andremo ad analizzare è riferita ad un percorso base in cui l'auto, dopo una fase iniziale di accelerazione, incontra una curva di 90° e la affronta a velocità costante. La prova ha una durata di un minuto circa e la velocità rimane costante dopo la fase di accelerazione.

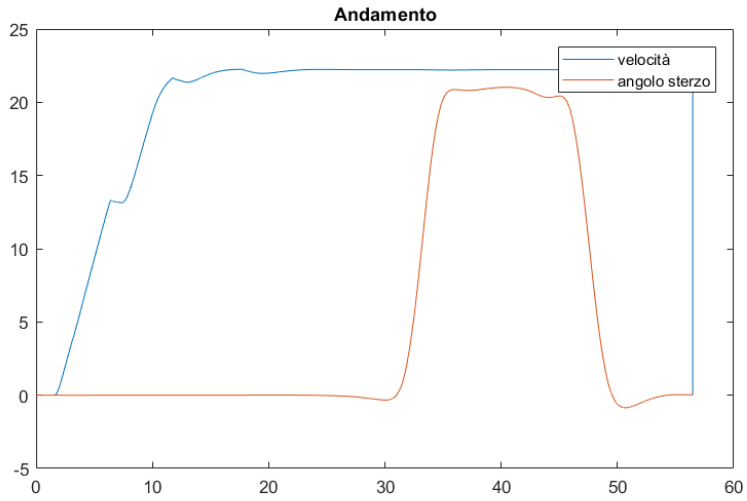


Figura 7.2. Il grafico mostra l'andamento dell'angolo sterzo e della velocità rispetto al tempo.

Dal successivo grafico si evince come il primo filtro di *Kalman* per la stima della u_n risulti essere molto preciso in tali condizioni; difatti la differenza tra i valori misurati sul simulatore e quelli calcolati dal filtro variano per cifre non troppo significative:

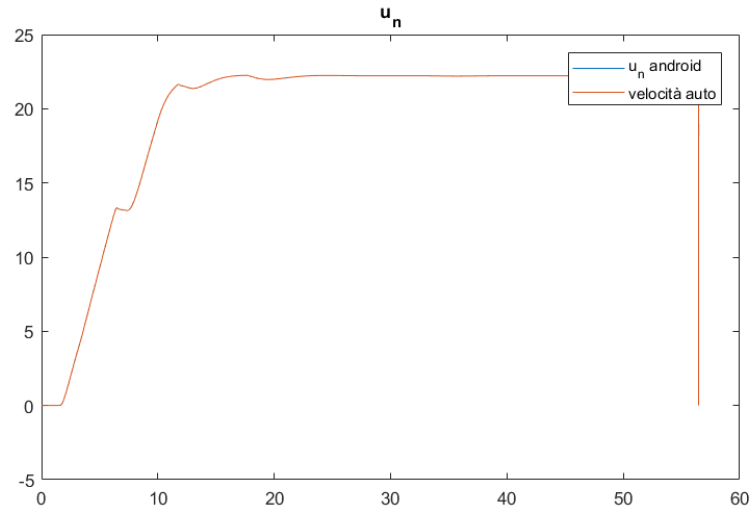


Figura 7.3. Confronto tra velocità del veicolo misurata e u_n stimata dal primo filtro di $Kalman$.

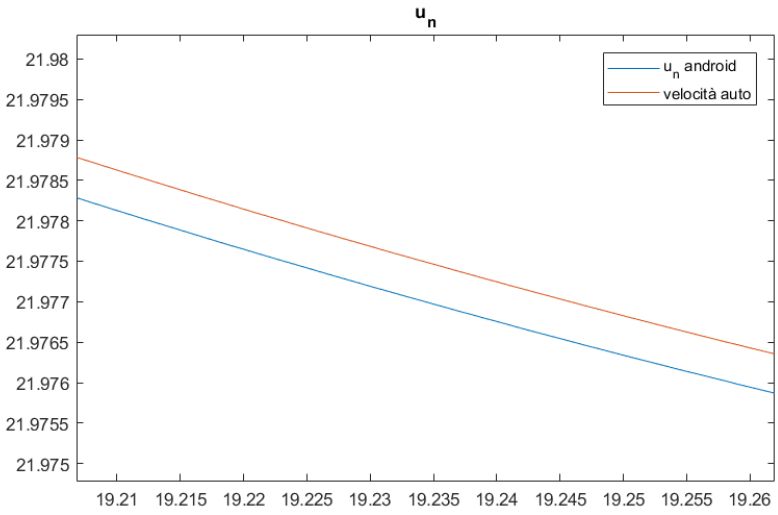


Figura 7.4. Dettaglio che mostra la differenza di valori tra il valore misurato e stimato.

Partendo quindi da una misura approssimata in modo ottimo il secondo filtro di $Kalman$ ci restituisce l'angolo di assetto β: dal grafico seguente si evince come il valore sia approssimato, istante per istante, in modo più che sufficiente poiché l'ordine di grandezza è comunque rispettato, il valore risulta diverso ma tale differenza è dovuta alle premesse fatte all'inizio di questo capitolo.

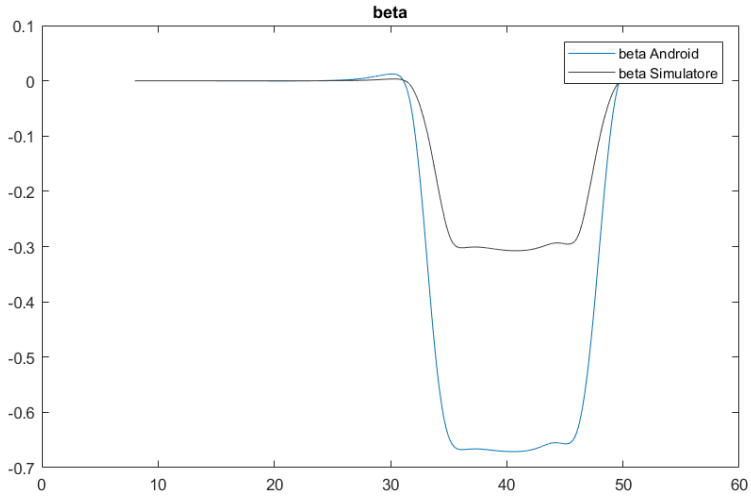


Figura 7.5. Andamento del β: confronto tra il valore calcolato dal simulatore e quello dell'algoritmo Android.

Il risultato più sorprendente risulta però essere quello seguente: confrontando i valori ottenuti con l'algoritmo implementato in MatLab, la misurazione sullo smartphone risulta praticamente identica: il fatto che il calcolo, pur avvenendo in modo diversificato e su due piattaforme diverse, risulti comunque essere congruente sta a significare che l'algoritmo sviluppato inizialmente ([**?**]) è molto valido dal punto di vista matematico e della portabilità.

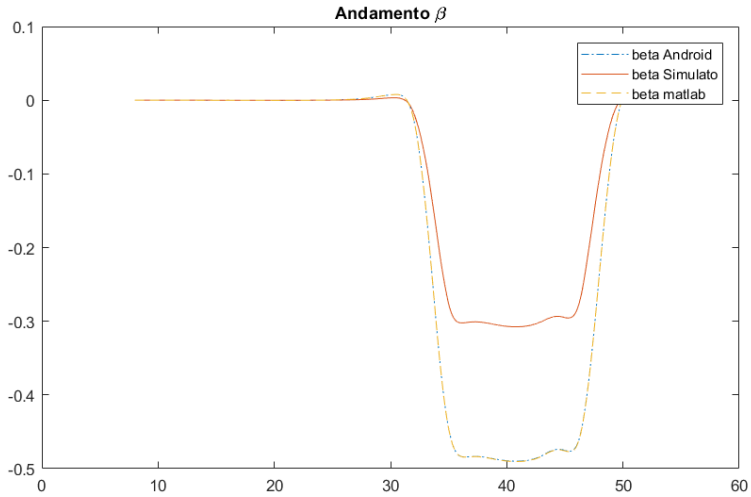


Figura 7.6. Confronto tra misurazioni del β ottenute dal simulatore, dallo smartphone e dall'algoritmo scritto in MatLab.

7.1.2 Prova 2: Curve Complesse

La seconda prova simulata è, come impostazione, molto simile alla prima ma è resa più *complicata* dalla tipologia di curve: come si evince dal grafico infatti il veicolo accelera, mantiene costante la velocità ed affronta tre curve significative; dopo mantiene la direzione rettilinea ed al termine affronta le ultime due curve, sempre a velocità costante. La durata della prova è all'incirca di quattro minuti (250 secondi).

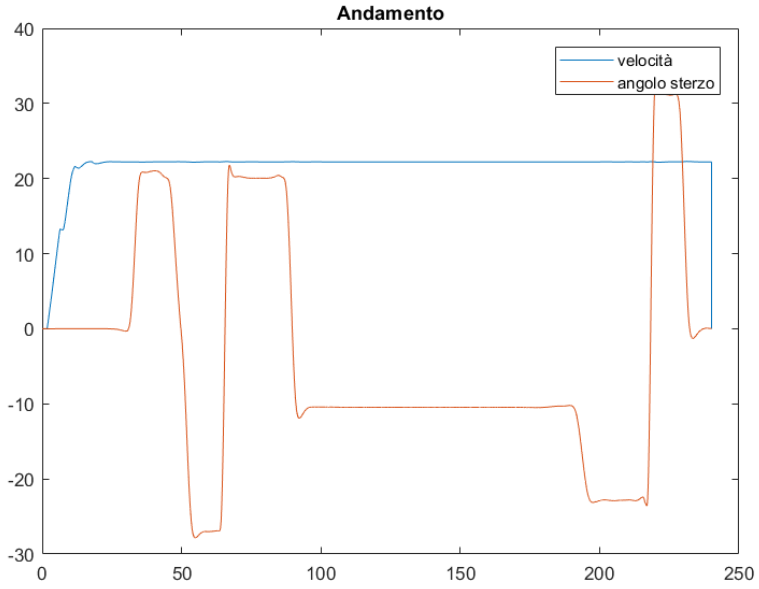


Figura 7.7. Andamento dell'angolo volante e della velocità del veicolo rispetto al tempo.

Anche in questo caso il primo filtro di *Kalman* agisce in maniera egregia calcolando l'approssimazione u_n in modo molto preciso, similmente a quanto visto per la prima prova:

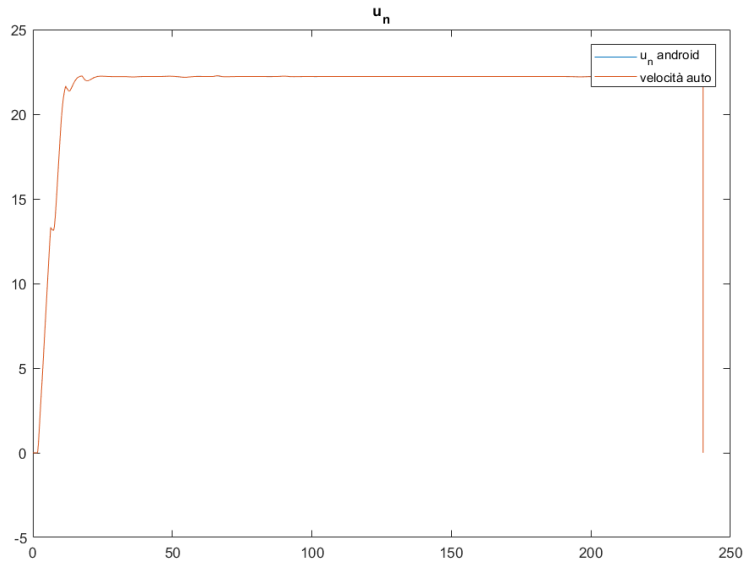


Figura 7.8. Confronto tra velocità del veicolo misurata e u_n stimata dal primo filtro di *Kalman*.

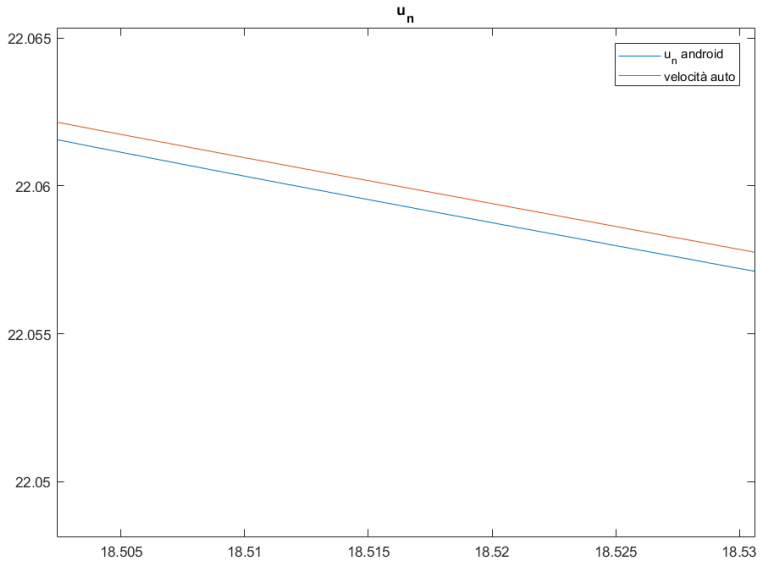


Figura 7.9. Dettaglio che mostra la differenza di valori tra il valore misurato e stimato.

Andando ad analizzare i risultati finali si evince come anche in questo caso l'algoritmo svolge il suo lavoro in maniera egregia, il valore calcolato infatti, è molto vicino al valore

stimato dal simulatore:

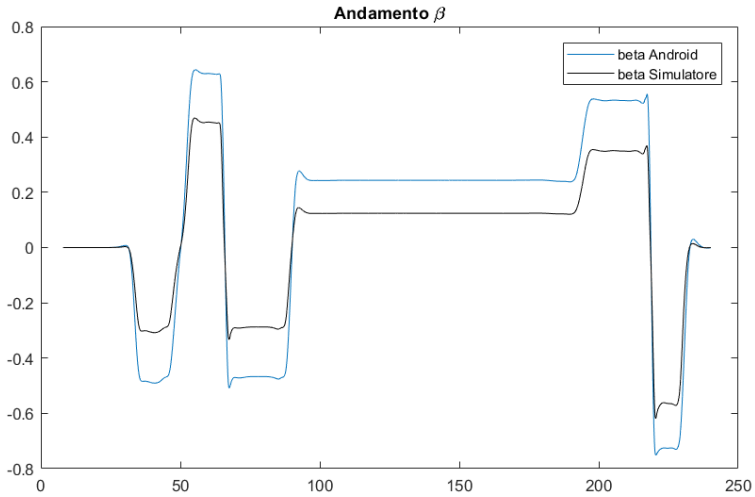


Figura 7.10. Andamento del β: confronto tra il valore calcolato dal simulatore e quello dell'algoritmo Android.

Per quanto riguarda il confronto con il valore calcolato da MatLab anche in questo caso la differenza di valore è pressoché nulla:

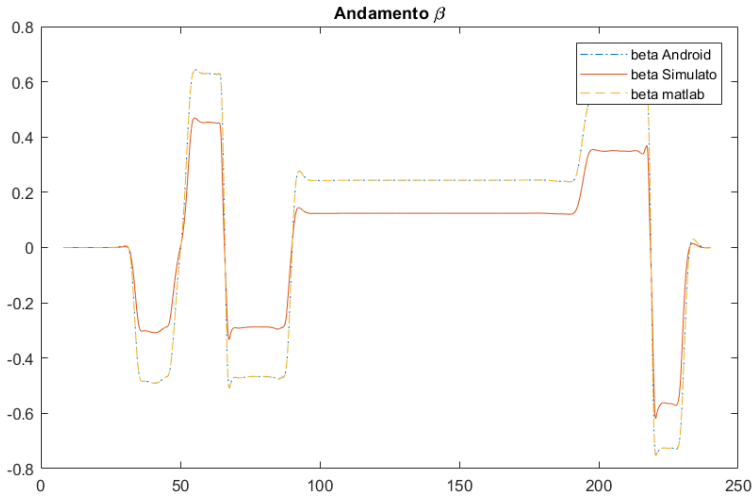


Figura 7.11. Confronto tra misurazioni del β ottenute dal simulatore, dallo smartphone e dall'algoritmo scritto in MatLab.

7.1.3 Prova 3: Veicolo su percorso Sterrato

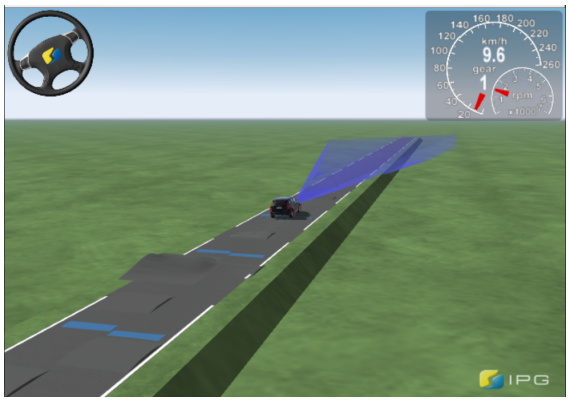


Figura 7.12. Vista ad un generico istante de software CarMaker.

La prova che segue è stata volontariamente eseguita in un contesto *estremo*: il veicolo utilizzato è un SUV Volvo XC90 ed il terreno presenta delle criticità durante tutto il percorso. Valutando difatti i valori delle accelerazioni e dell'angolo volante si nota come i dati risultano particolarmente disturbati dall'andatura sul percorso. La prova è stata effettuata per valutare la robustezza dell'algoritmo nei casi estremi; tuttavia c'è da considerare che una prova del genere non sarebbe stata riproducibile nella realtà per problemi con il mantenimento delle informazioni relative all'angolo volante.

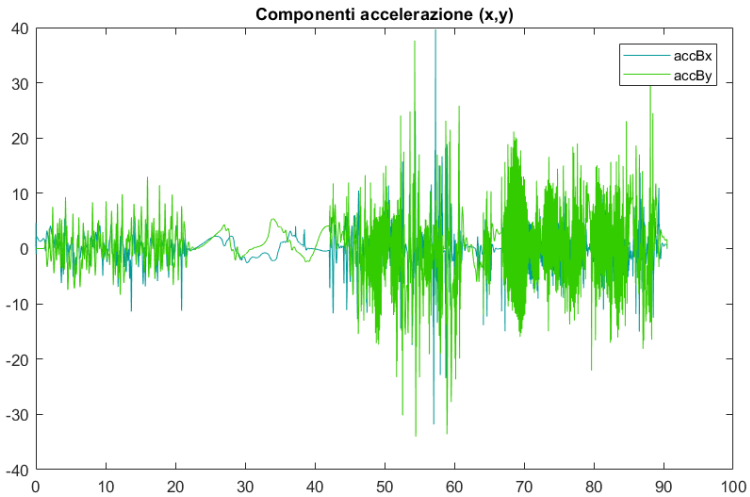


Figura 7.13. Andamento delle componenti (x, y) di accelerazione rispetto al tempo.

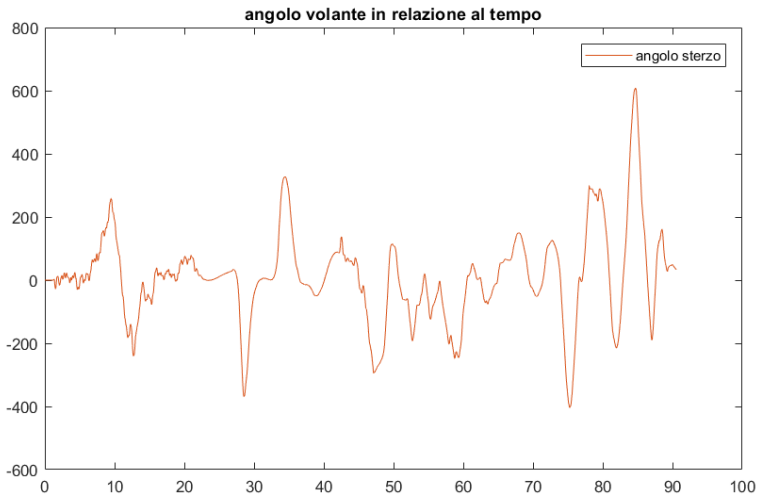


Figura 7.14. Andamento dell'angolo volante rispetto al tempo.

Confrontando il grafico relativo all'angolo di assetto si nota come, anche nel caso del simulatore, i valori siano per certi versi *insensati*: l'angolo di assetto infatti dovrebbe variare al massimo nell'ordine di 5-6° mentre in questo caso abbiamo variazione dell'ordine di 10° inizialmente ed addirittura un picco negativo a circa -40°! Ciò dimostra che la prova sia in ogni caso non riproducibile. Ponendo attenzione però sul confronto tra il β stimato dall'applicazione e quello del software di simulazione si nota come nonostante l'andamento inaspettato questo sia comunque in linea con quello calcolato: per certi istanti di tempo i valori sembrano del tutto insensati (addirittura il segno risulta cambiato) ma aldilà di ciò l'andamento generale viene comunque rispettato.

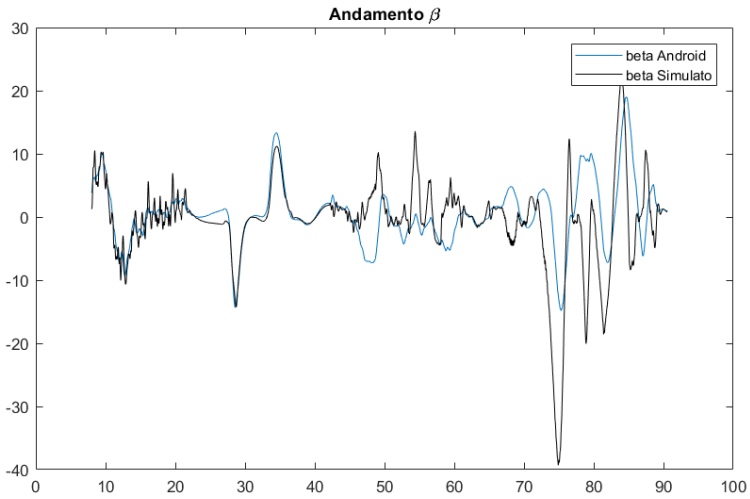


Figura 7.15. Andamento dell'angolo di assetto β rispetto al tempo.

7.1.4 Prova 4: Veicolo su Pista

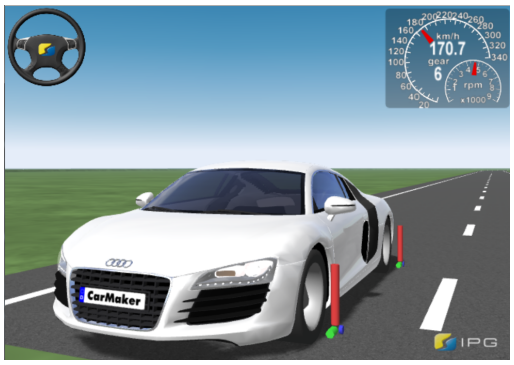


Figura 7.16. Veicolo utilizzato durante la prova su pista.

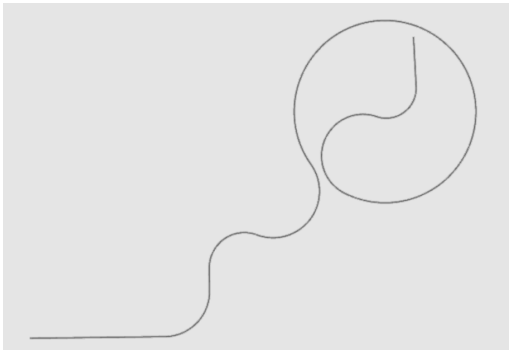


Figura 7.17. Mappatura del percorso effettuato.

In questa ultima prova simulata è stato utilizzato un pilota simulato con guida altamente *sportiva*: le velocità raggiunte e le curve sono infatti al limite del fattibile nella realtà. Tuttavia è interessante notare il confronto tra i valori calcolati dal simulatore e quelli stimati dall'applicazione Android.

Analizzando il grafico della velocità e dell'angolo sterzo rispetto al tempo si nota come, dopo una fase iniziale di velocità in rettilineo il veicolo affronti tre rapide curve ed infine una *rotonda* a velocità incrementale.

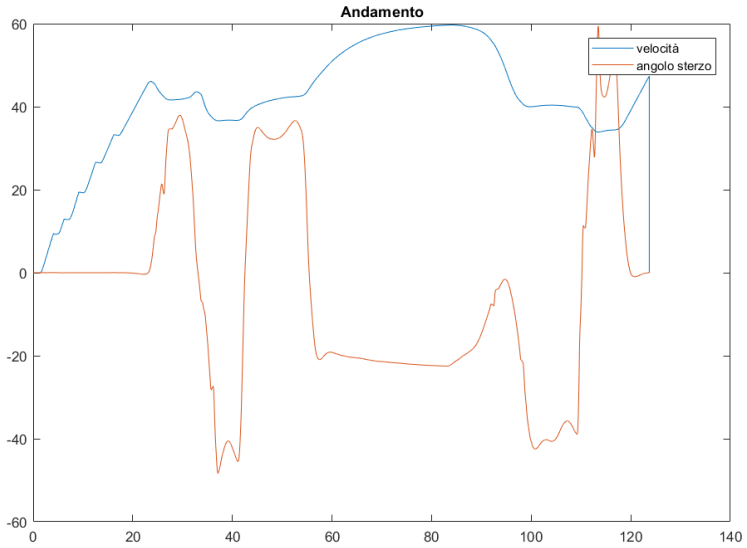


Figura 7.18. Andamento della velocità e dell'angolo sterzo rispetto al tempo.

Confrontando invece i grafici del β si nota come, nonostante la guida *sportiva* i valori non si discostino molto dalla controparte misurata: la fase *critica* di stima risulta essere infatti l'accelerazione in curva da parte del veicolo e la successiva accelerazione in uscita dalla stessa. In tal caso il valore del β stimato infatti tende quasi ad azzerarsi mentre quello misurato mantiene un angolazione di circa 2,5°.

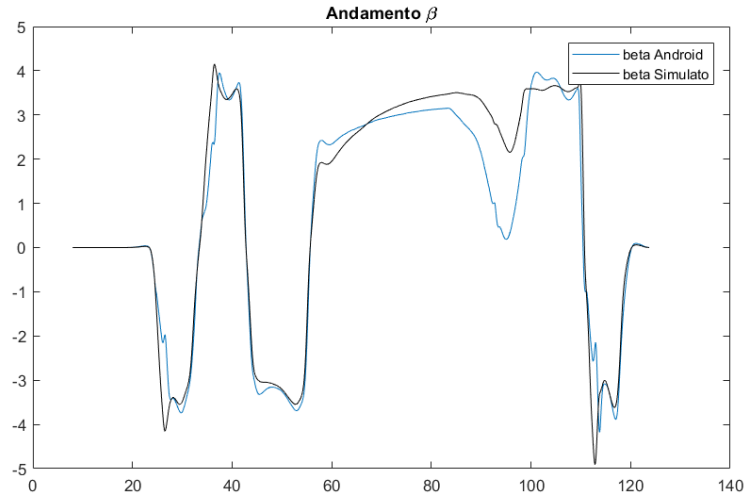


Figura 7.19. Andamento del β rispetto al tempo.

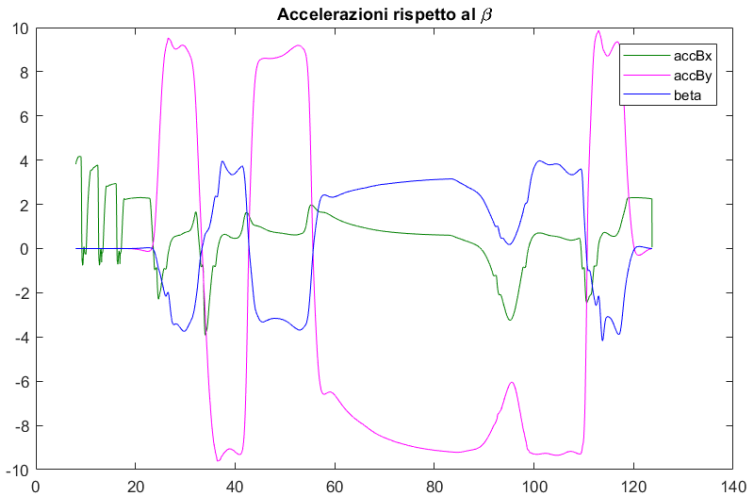


Figura 7.20. Andamento del β e delle componenti (x, y) di accelerazione rispetto al tempo.

7.1.5 Conclusioni

Tramite il confronto con le prove simulate su *CarMaker* si è reso noto come l'algoritmo, nonostante lavorasse in condizioni *ideali* (acquisendo e mantenendo quindi l'angolo sterzo, ricevendo i dati dai sensori ad una frequenza più bassa) riesce comunque a mantenere la robustezza ipotizzata inizialmente e in certi frangenti risulta addirittura più performante e preciso rispetto alla controparte sviluppata inizialmente in MatLab. Detto ciò ovviamente l'algoritmo è sviluppato sulle *fondamenta* della versione MatLab quindi mantiene comunque gli stessi limiti conosciuti inizialmente e ciò si vede infatti nella prova su percorso sterrato (7.1.3).

7.2 Prove su Strada

7.2.1 Prova 1

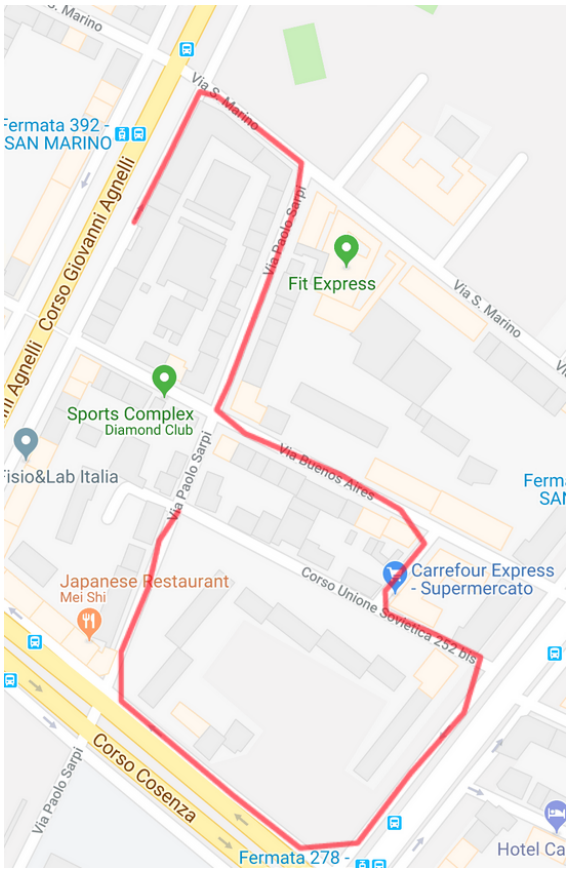


Figura 7.21. Percorso prova 1, vista da Google Maps.

La prima prova su strada è stata effettuata su un percorso urbano di Torino a bordo di una Alfa Mito (segmento C) con uno stile di guida da città: come si nota dai grafici la velocità è comunque moderata, con picchi nei percorsi rettilinei. L'angolo sterzo invece varia di molto sopratutto in prossimità delle curve *a gomito*, tipiche dei percorsi cittadini. Per la visualizzazione dei dati questi sono stati esportati in un file CSV ed elaborati su MatLab:

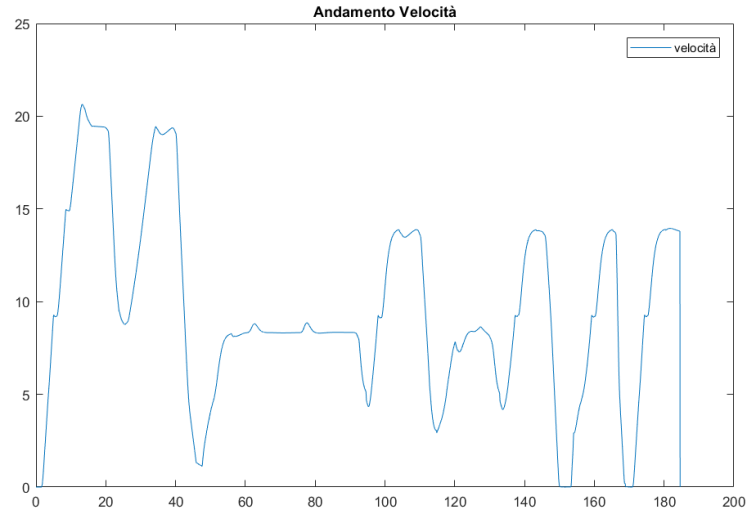


Figura 7.22. Andamento della velocità del veicolo durante la prova.

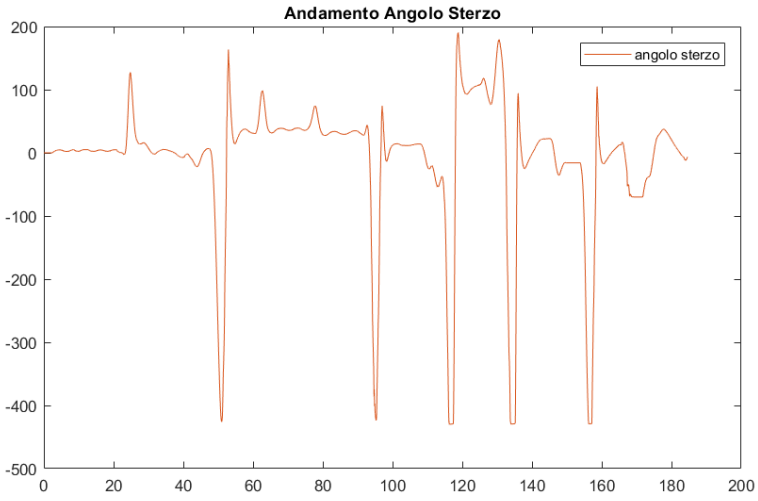


Figura 7.23. Andamento dell'angolo volante durante la prova.

Per quanto riguarda invece i risultati del β: si nota come, sopratutto nella fase iniziale, l'algoritmo si comporti in maniera accettabile, a fronte comunque di bruschi cambi di direzione. Nella parte finale invece vi è un andamento non tanto in linea con il valore reale. Risultano infatti esserci, a fronte di valori specifici, delle *singolarità* nel calcolo del β:

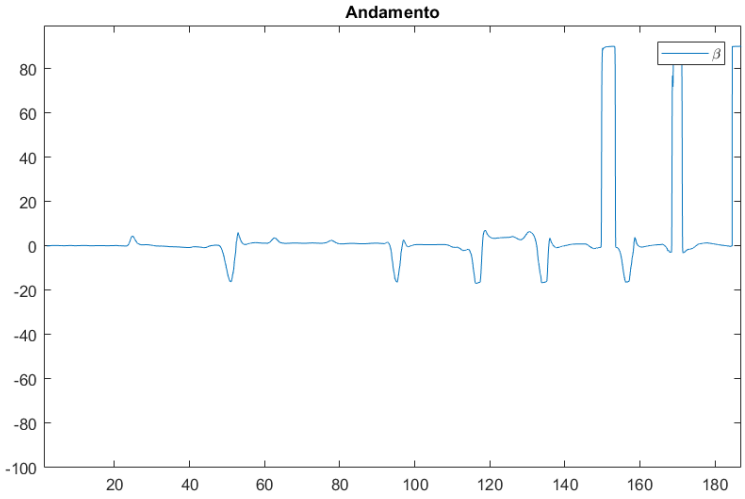


Figura 7.24. Andamento del β.

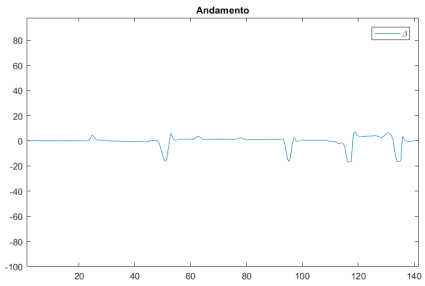


Figura 7.25. Dettaglio rispetto alla prima parte della prova.

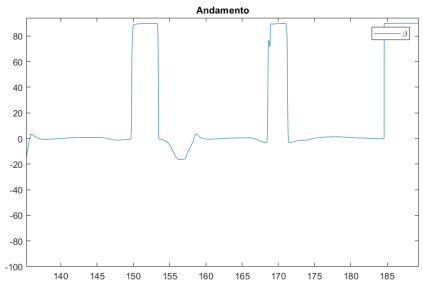


Figura 7.26. Dettaglio rispetto alla seconda parte della prova.

Andando ad analizzare il grafico di β in relazione alla velocità risulta esserci una corrispondenza tra i valori a *zero* della componente V ed il valore della singolarità nel calcolo del β:

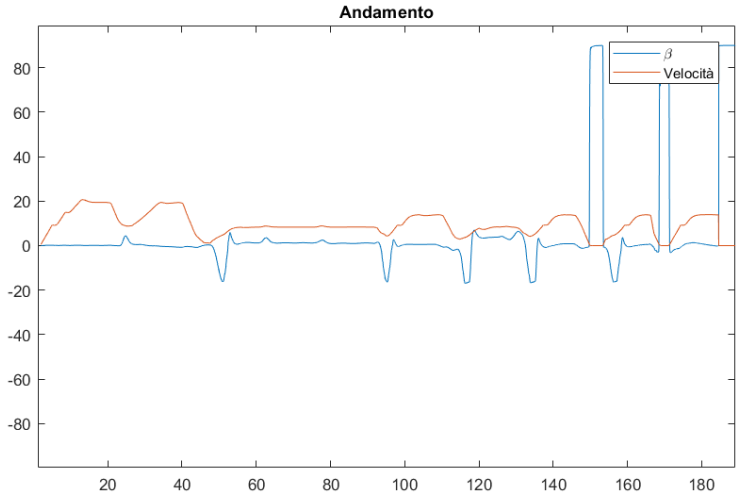


Figura 7.27. andamento di V e β rispetto al tempo.

Vedendo nel dettaglio la parte relativa alla seconda metà della prova, risulta evidente che l'andamento del β dipenda da V.

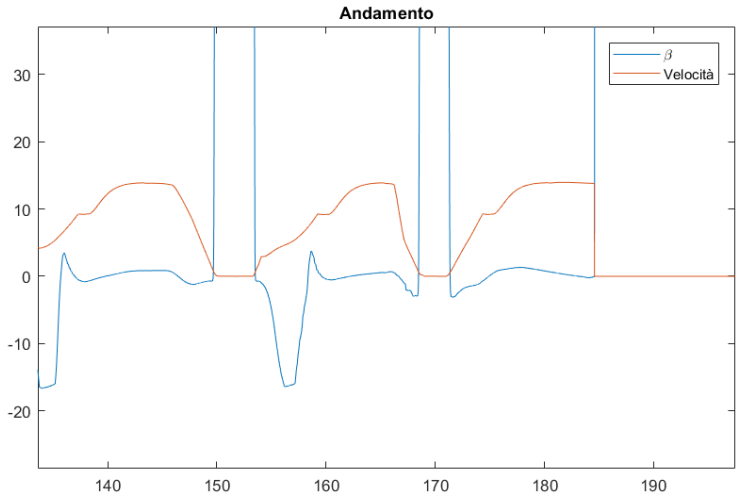


Figura 7.28. Dettaglio: andamento di V e β rispetto al tempo.

Analizzando in dettaglio gli istanti in cui il calcolo del β rasenta valori *enormi*, si nota come questo sia legato al fatto che la componente V è *nulla*: ciò dimostra il fatto che, operando in post-processing o comunque modificando l'algoritmo in modo tale da evitare tale situazione, si possa venir meno a situazioni di questo tipo. Aldilà di tale situazione il calcolo risulta comunque essere molto promettente.

Di seguito alcuni screen dei grafici rappresentati dall'applicazione stessa: vengono selezionati dalla mappa particolari punti di cui si vuole vedere l'andamento.

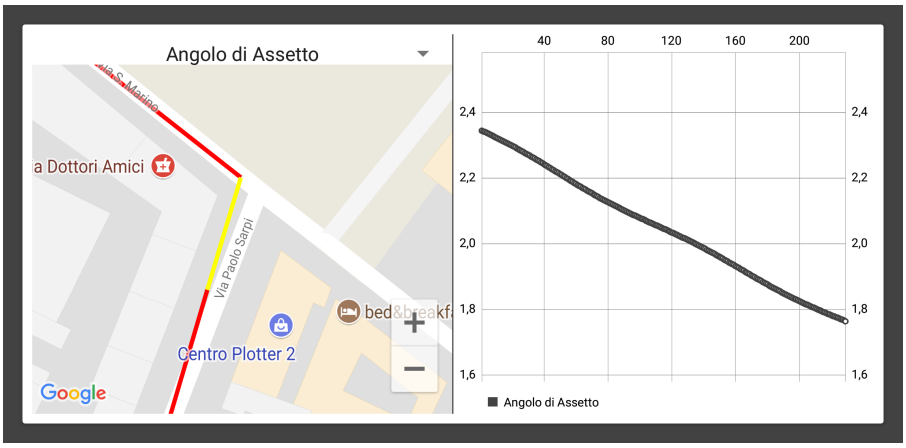


Figura 7.29. Dettaglio: vista del grafico di β all'interno dell'applicazione.

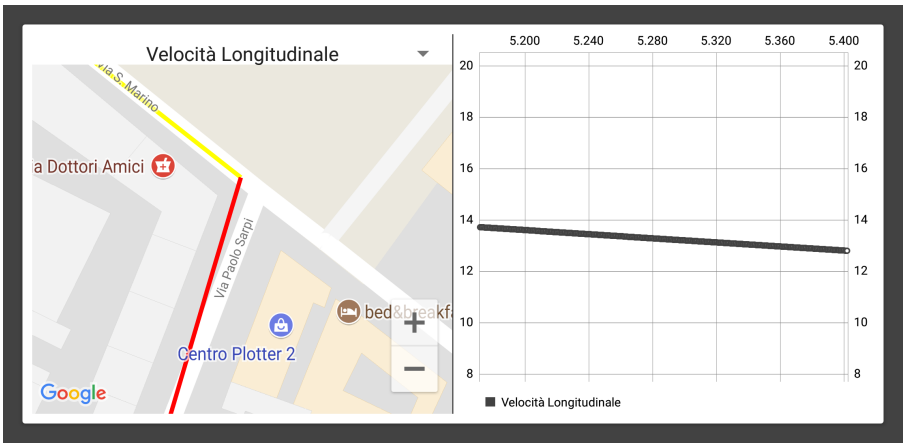


Figura 7.30. Dettaglio: vista del grafico di v all'interno dell'applicazione.

Figura 7.31. Dettaglio: vista del grafico di δ_v all'interno dell'applicazione.

7.2.2 Prova 2

La seconda prova su strada è stata effettuata a Modica, un paesino a sud della Sicilia, a bordo di una Fiat Punto (Segmento B). Il percorso è molto breve ma molto significativo poiché, come si evince dalla mappatura, comprende sia una rotonda che una curva ad U che un tratto rettilineo di accelerazione. Anche in questo caso i dati sono stati raccolti dalla sensoristica dello smartphone e successivamente esportati ed elaborati su MatLab. C'è da considerare il fatto che, a differenza della prima prova, questa volta erano presenti altri veicoli e quindi in certe situazioni noteremo una velocità pari a zero a causa delle soste provvisorie causa traffico. Per quanto riguarda l'angolo volante invece, la prima depressione rappresenta la curva a destra all'inizio della prova, mentre la seconda forte depressione rappresenta la manovra ad U effettuata successivamente: in questo caso infatti l'angolo sterzo è stato *perso* per gli istanti in cui le braccia erano davanti ai markers. C'è da dire però che sono stati recuperati in modo quasi perfetto e riportati a zero dopo la manovra. Gli ultimi istanti invece, a velocità nulla, rappresentano i momenti finali in cui lo sterzo ruotava per il parcheggio; quindi potrebbero essere tranquillamente ignorati.

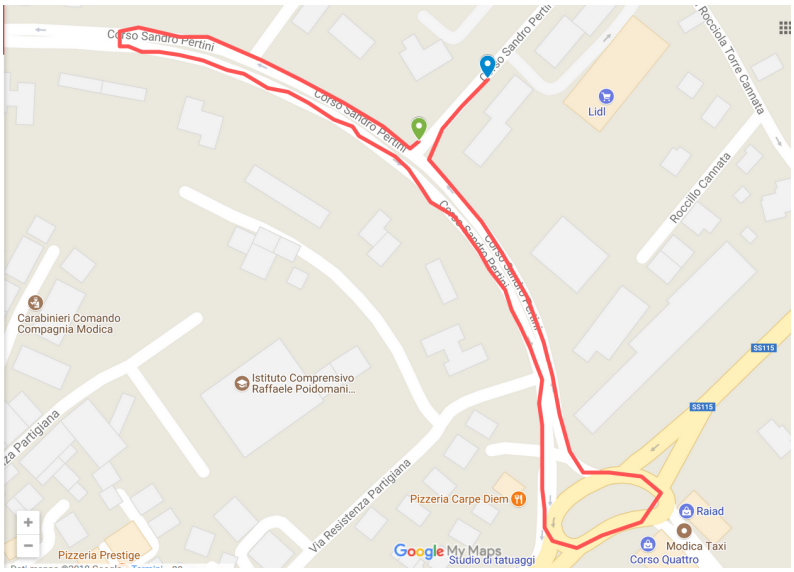


Figura 7.32. Percorso prova 2, vista da Google Maps.

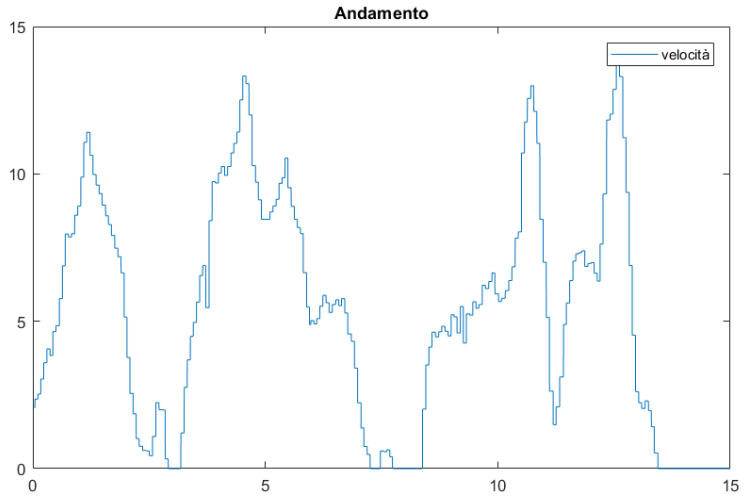


Figura 7.33. Andamento della velocità del veicolo durante la prova.

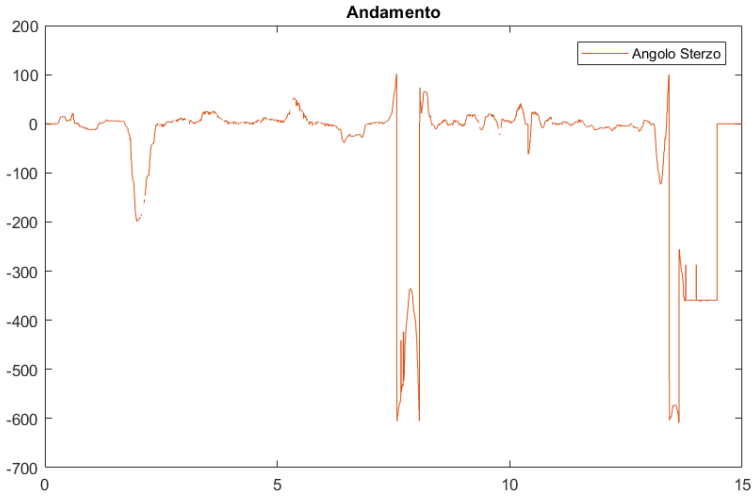


Figura 7.34. Andamento dell'angolo volante durante la prova.

Per quanto riguarda i risultati inerenti al β invece si nota come risultino molto fedeli al percorso effettuato:

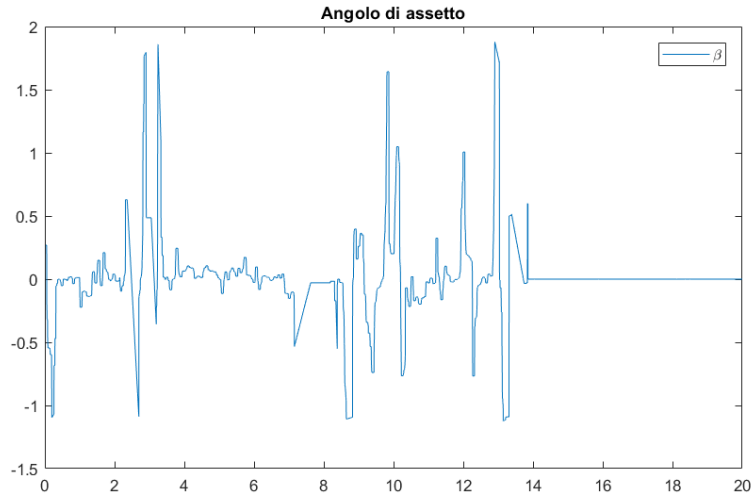


Figura 7.35. Andamento dell'angolo d'assetto durante la prova.

In particolare andando ad analizzare il grafico rispetto a quello della velocità si nota come vi siano alcune criticità in corrispondenza di misurazioni effettuate a veicolo fermo: infatti, essendo la velocità prossima allo zero, il rapporto di alcuni valori risulta essere anomalo. Tale comportamento è facilmente arginabile tramite elaborazione in *post* sul grafico o ignorando le misurazioni in corrispondenza di velocità nulle.

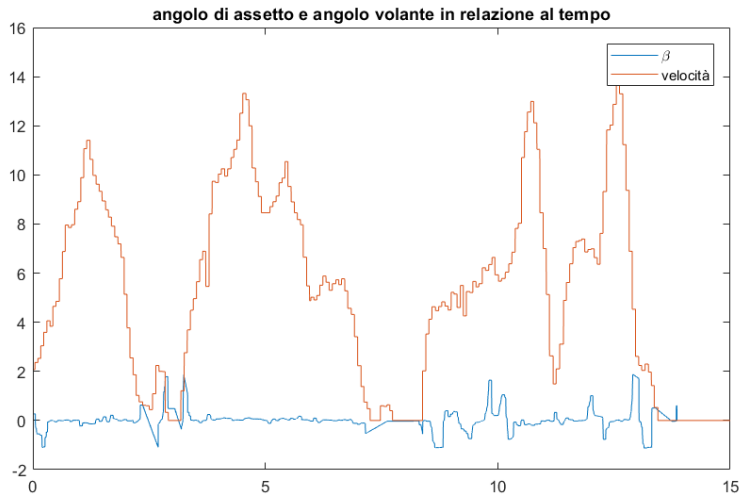


Figura 7.36. Andamento dell'angolo d'assetto e della velocità durante la prova.

Di seguito gli screenshot relativi alla prova, visti direttamente all'interno dell'applicazione:

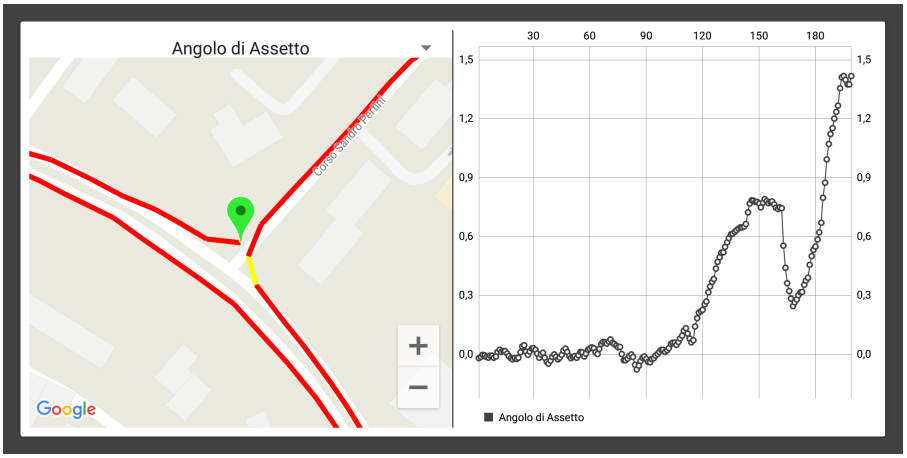


Figura 7.37. Dettaglio: vista del grafico di β all'interno dell'applicazione.

Figura 7.38. Dettaglio: vista del grafico di δ_v all'interno dell'applicazione.

Figura 7.39. Dettaglio: vista del grafico della velocità all'interno dell'applicazione.

Capitolo 8

Conclusioni

8.1 Considerazioni Personali

Il lavoro di tesi svolto è stato, personalmente, molto stimolante e competitivo sia dal punto di vista lavorativo, poiché molte tecnologie utilizzate e l'intera stesura del codice hanno presupposto uno studio più o meno dettagliato dell'argomento relativo alla dinamica del veicolo, all'interazione con la sensoristica dello smartphone e ad argomenti riguardanti il ramo di computer vision, sia dal punto di vista personale: essendo infatti il primo elaborato interamente sviluppato da me medesimo, ho avuto molte difficoltà nello scrivere ed elaborare la parte *pratica*, ma anche nel comprendere fenomeni del tutto sconosciuti fino a poco tempo fa. Consiglierei comunque ad un futuro ingegnere informatico di *buttarsi* su una tesi multi-disciplinare come la seguente poiché gli orizzonti dell'informatica in generale hanno bisogno di essere considerati anche sotto altri punti di vista e magari applicati in altri ambiti che non siano strettamente elettronici/informatici.

8.2 Miglioramenti Futuri

Per quanto riguarda i futuri *improvements* invece, il lavoro dovrà vertere sull'arginare i limiti che tutt'ora sono presenti sia negli algoritmo precedentemente sviluppati, come ad esempio l'adattamento delle matrici di covarianza nel calcolo del β, l'elaborazione parallela dei task che potrebbe essere ripensata in ambito $client - server$, ed anche alcune possibili vie sulla lettura dell'angolo volante: è stato sperimentato infatti una lettura tramite l'applicazione di *ARuco* Markers, dei particolari marker che consentono di essere riconosciuti anche in condizioni di illuminazione bassi poiché la libreria è stata sviluppata appositamente per il tracking di tali marker. In più è possibile elaborare la posizione nello spazio 3D dei marker in ogni singolo frame grazie al riconoscimento spaziale degli stessi: sono infatti molto utilizzati nell'ambito della *Realtà Aumentata* proprio perché permettono, dopo una calibrazione iniziale della camera, di essere riconosciuti all'interno dello spazio reale. In seguito magari si potrebbe anche pensare di eliminare del tutto i marker ed applicare degli algoritmi di *Machine Learning* direttamente per il riconoscimento del volante e della sua rotazione istante per istante: eseguendo un'elaborazione intensa infatti si potrebbe ovviare a molto dei problemi precedentemente esposti sulla lettura dell'angolo volante. Inoltre

l'applicazione potrebbe interagire con un server centrale per la raccolta ed elaborazione dati, così da non coinvolgere direttamente il terminale e poi mostrare, in post, i risultati calcolati ed archiviati direttamente sul server.

Parte IV

Fine

Bibliografia

[1] Stefano Delzoppo, *Rilevazione della dinamica di un veicolo attraverso l'impiego di una piattaforma inerziale*, Tesi Magistrale, Politecnico di Torino, 2015.

[2] Andrea Mucci, *Utilizzo di una videocamera ed una piattaforma inerziale low cost per il rilievo di grandezze di dinamica del veicolo*, Tesi Magistrale, Politecnico di Torino, 2016.

[3] Marco Albanese, *Stima dell'angolo di assetto di un veicolo*, Tesi Magistrale, Politecnico di Torino, 2017.

[4] Wikipedia, *Equazione di Ricciati*, `https://it.wikipedia.org/wiki/Equazione_di_Riccati`, 2016.

[5] A. Antonielli, G. D'Avico, M. Paciscopi, *Corso di Analisi di Immagini e Video*, Laurea Magistrale in Ingegneria Informatica, Università degli studi di Firenze, 2012.

[6] Amy Rose, *Procrustes Analysis*, Department of Computer Science and Engineering, University of South Carolina, 2002.

[7] Ricardo Tenreiro, *Android Overview*, `http://elearning.rtenreirosa.info/en/android`, 2016.

[8] Alberto Gatto, *Tracking in Android per NXT*, Tesi di Laurea in Ingegneria Informatica, Università degli studi di Padova, 2013.

[9] G. Malnati, C. Barberis, *Mobile Application Development*, Corso di Laurea Magistra in Ingegneria Informatica, Politecnico di Torino, 2017.

[10] Xavier Hallade, *Using Native Development Kit*, `https://www.slideshare.net/ph0b/using-the-android-native-development-kit-ndk`, 2014.

[11] Di Ruberto, Cecilia & Putzu, Lorenzo, *A fast leaf recognition algorithm based on SVM classifier and high dimensional feature vector.* VISAPP 2014.

[12] Wikibooks, *Color Models: RGB, HSV, HSL* `https://en.wikibooks.org/wiki/Color_Models:_RGB,_HSV,_HSL`, 2017.

[13] Ornella Di Scala, *Dipinto quadro moderno Pulcinella pop art dipinto a mano Napoli rosso arredo* `https://www.pitturiamo.com/it/quadro-moderno/dipinto-quadro-moderno-pulcinella-pop-art-dipinto...`, 2015.

[14] OpenCv, *OpenCv Documentation* `https://docs.opencv.org/3.3.1/index.html`, 2017.

[15] OpenCv, *Moments Class Reference* `https://docs.opencv.org/3.4.0/d8/d23/classcv_1_1Moments.html`, 2017.

[16] Wikipedia, *Singular-value decomposition*, `https://en.wikipedia.org/wiki/Singular-value_decomposition`, 2018.

[17] Nghia Ho, *Finding Optimal Rotation and Translation between corresponding 3D Points* http://nghiaho.com/?page_id=671, 2013.

[18] Rakesh Singh, *Explain Java BlockingQueue with Producer-Consumer thread example* http://www.interviewsansar.com/2014/09/20/java-blockingqueue-example-with-produce-and-consumer-design-pattern/, 2014.

[19] mrmans0n, *Smart Location Library* https://github.com/mrmans0n/smart-location-lib, 2017.

[20] lessthanoptimal, *GeoRegression Library* https://github.com/lessthanoptimal/GeoRegression, 2018.