ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DIPARTIMENTO DI INGEGNERIA INDUSTRIALE (DIN)

CORSO DI LAUREA IN
INGEGNERIA AEROSPAZIALE

SVILUPPO E PROTOTIPAZIONE DI UN SISTEMA DI AUSILIO ALLA NAVIGAZIONE AEREA SU PIATTAFORMA MOBILE

Tesi di laurea in Meccanica del volo

Relatore

Prof. Emanuele Luigi de Angelis

Presentata da:

Andrea Calizzani

Sessione ottobre 2024

Anno Accademico 2023/2024

SOMMARIO

Nel presente lavoro è stata sviluppata un'applicazione per dispositivi mobili in grado di supportare i piloti di elicotteri nell'inseguimento di una traiettoria desiderata, secondo due diverse modalità: una che conduce direttamente al target (obiettivo), l'altra che permette di inserirsi in una traiettoria definita "ottimale", di collegamento tra un punto iniziale e un punto finale. Questo inserimento è possibile grazie ad un'interfaccia grafica appositamente sviluppata in maniera tale che risulti il più intuitiva e di facile fruizione, consentendo al pilota di correggere l'angolo di rotta nella modalità più rapida ed efficace possibile.

Il presente lavoro si focalizza sulla progettazione dell'interfaccia grafica e sull'implementazione dell'algoritmo di guida, garantendo un'esperienza d'uso ottimale per il pilota.

INDICE

INDICE DELLE FIGURE

INDICE DELLE TABELLE

1. INTRODUZIONE

Sulla superficie terrestre, per spostarsi da un punto ad un altro, vengono seguite delle indicazioni che si basano su riferimenti materiali e precisi come semafori, incroci o rotonde; inoltre vista la presenza delle varie infrastrutture, non è possibile seguire un percorso in linea retta dal punto iniziale a quello finale. In aeronautica, invece, è possibile seguire percorsi in linea retta ma non esistono riferimenti fisici precisi. Un metodo è quello di definire arbitrariamente dei punti tramite delle coordinate GPS (chiamati waypoint [1]) tali che il passaggio su ognuno di questi punti, con la giusta sequenza, determini la rotta dell'aeromobile.

Il supporto al raggiungimento di un waypoint desiderato è l'intuizione dalla quale nasce questa applicazione, ovvero essere di ausilio ad un pilota di elicotteri nel raggiungere un obiettivo (detto anche target) tramite un'interfaccia grafica che rappresenti l'errore tra l'angolo di rotta istantaneo e quello ottimale per raggiungere tale punto [2].

Si ritiene che questa applicazione possa essere un valido supporto soprattutto ai piloti di piccoli elicotteri in quanto i sistemi di bordo installati su di essi non sono così avanzati come in quelli di maggiori dimensioni. Infatti, negli elicotteri più avanzati, è possibile inserire all'interno del computer di bordo, l'intera rotta definita da una serie di waypoint, inseguiti uno alla volta. È importante specificare che normalmente il velivolo non segue automaticamente tutti i waypoint ma è sempre manovrato dal pilota che adotta le indicazioni fornite dal computer di bordo. Le informazioni gli vengono fornite principalmente in maniera grafica, infatti si possono trovare rappresentazioni cartografiche in cui viene indicata la posizione del target e quella del velivolo oppure anche solamente una rappresentazione della differenza tra la rotta istantanea e quella ottimale (come avviene nell'applicazione oggetto di questa relazione). In alternativa però, si possono seguire alcune strumentazioni o delle rappresentazioni simboliche (come delle frecce) che rappresentano la direzione in cui si trova la destinazione. In particolare, quest'ultima tipologia è molto comune in quelli che vengono definiti head-up display ovvero dei dispositivi trasparenti posti

davanti agli occhi del pilota in cui vengono rappresentati i dati di volo affinché possano essere monitorati senza perdere il contatto visivo con l'orizzonte [3].

Riferendosi nuovamente ai velivoli più leggeri e meno avanzati in quanto presentano una dotazione sistemistica di minor complessità, non è raro notare che i piloti di tali velivoli utilizzano spesso dispositivi digitali personali come tablet o smartphone a supporto della navigazione. Tale fenomeno si verifica in quanto, all'interno di tali dispositivi, è possibile installare applicativi che uniscono in un unico portale la cartografia, alcune strumentazioni di bordo, i navigatori e forniscono inoltre avvertenze in caso di violazioni di normative. Queste applicazioni appartengono alla categoria delle EFB ovvero "electronic flight bag" [4]. Tale soluzione si rivela certamente vantaggiosa, presentando un costo significativamente inferiore rispetto a un sistema avionico fisso installato all'interno della cabina di pilotaggio. Quest'ultimo, tuttavia, offre il beneficio di un'integrazione più completa con altri sistemi di bordo, permettendo così una maggiore accuratezza nella raccolta dei dati.

Preso atto che i dispositivi personali sono comunque molto utilizzati in questa particolare categoria di velivoli, si è pensato di sviluppare un'applicazione specifica per supportare i piloti di elicotteri nel raggiungere un determinato waypoint. In particolare, definendo un punto di partenza e uno di arrivo e collegandoli attraverso una linea retta che rappresenta il percorso ottimale e teoricamente percorribile, si intende che, data una qualunque posizione del velivolo, questo possa raggiungere l'obiettivo in una delle due modalità implementate. Più precisamente, il pilota potrà scegliere se raggiungere il target seguendo una traiettoria lineare che conduce direttamente all'obiettivo oppure se utilizzare il beam rider ovvero una modalità che gli consente di immettersi al meglio nella rotta ottimale e di persistere su di essa, correggendo anche i piccoli errori che si possono generare durante il volo, istante per istante.

L'obiettivo dell'attività di tirocinio descritta è quello di sviluppare un'interfaccia grafica (GUI- Graphical User Interface) in grado di supportare il pilota nel permanere nella rotta corretta con particolare attenzione all'intuitività con cui vengono riportate le informazioni [5].

Il metodo sviluppato, considerato come il più immediato possibile per il pilota, è quello di un cerchio che si muove su di un piano cartesiano e la cui posizione rappresenta l'errore tra la direzione del velivolo e quella di riferimento della rotta calcolata. Si comprende quindi che lo scopo del pilota è quello di condurre il cerchio all'origine del sistema, annullando l'errore di rotta attraverso le opportune manovre [6]. Più precisamente, il cerchio rappresenta la direzione esatta mentre il centro del sistema indica la rotta del velivolo. La maggior parte della visualizzazione è occupata da questa grafica mentre, nella parte restante, è stata inserita una rappresentazione della direzione della velocità rispetto ai punti cardinali, nonché alcuni dati di volo come la differenza di velocità tra quella richiesta per il passaggio sull'obiettivo e quella attuale, l'angolo di rotta istantaneo, la velocità, la distanza dal target e la differenza di altitudine tra il target e l'altitudine attuale.

L'attività di tirocinio è stata suddivisa nelle fasi di seguito descritte:

- la prima fase, in cui è stata ricercata la migliore soluzione per realizzare l'attività assegnata, con un'attenzione particolare a quella che è la potenzialità del software. Si ritiene importante specificare che l'uso di applicazioni mobile è solo ai fini di una prototipazione rapida in quanto tali dispositivi contengono una buona dotazione sensoristica interna e quindi è necessario implementare solamente il software. Per la produzione finale sono invece necessari sensori e hardware di tipo industriale al fine di garantire precisione e soprattutto l'affidabilità del sistema. Dalla prima fase è scaturita la scelta del software Android Studio come base per la prototipazione vista la sua grande flessibilità e intuitività;
- la seconda fase, durante la quale sono state comprese le basi del linguaggio Java ed è stata acquisita familiarità con il programma, arrivando a sviluppare quello che è il nucleo centrale dell'applicazione stessa, ossia la visualizzazione della pallina;
- la terza fase ha riguardato la georeferenziazione del dispositivo ovvero sono state ricavate le coordinate GPS, sono state convertite nel sistema NED e successivamente le stesse sono state utilizzate per implementare l'algoritmo per l'inseguimento del target. L'output dell'algoritmo, ovvero l'errore di rotta, è

stato poi fornito alla sezione che si occupa della visualizzazione grafica per la rappresentazione;

- la quarta fase, durante la quale è stata sviluppata la parte restante della visualizzazione, inserendo alcuni dati relativi al moto del velivolo;
- la quinta fase, in cui è stato rivisto l'intero progetto dal punto di vista grafico, implementando inoltre l'errore di velocità e un filtro per stabilizzare i dati prodotti dall'unità posizionale.

Per una presentazione ottimale del lavoro svolto, questa relazione è stata suddivisa in cinque capitoli (di cui uno introduttivo), nei quali ne sono stati analizzati gli aspetti principali. In particolare, nel capitolo 2 è stato illustrato l'ambiente di sviluppo adottato, Android Studio, con un approfondimento sulle sue caratteristiche e funzionalità. Successivamente, nel capitolo 3, è stato esaminato l'algoritmo di navigazione, dal quale viene generato il dato che alimenta la visualizzazione principale. Nel capitolo 4, è stato descritto in dettaglio il codice dell'applicazione, suddiviso per le sue sezioni. Infine, nel capitolo 5 sono state tratte le conclusioni sul lavoro svolto e sono state fornite alcune indicazioni riguardo a possibili sviluppi futuri.

2. AMBIENTE DI SVILUPPO

L'ambiente di sviluppo scelto per questa prototipazione è Android Studio, un software prodotto da Google che permette di sviluppare applicazioni per dispositivi che utilizzano un sistema operativo di tipo Android come smartphone o tablet, ma potenzialmente anche per dispositivi indossabili (Wear-OS), Android-TV, Android Auto e Android Things. È un programma molto potente, intuitivo e soprattutto completo in quanto permette di gestire in un unico portale ogni aspetto relativo alla creazione di un'applicazione come la parte di programmazione effettiva, lo sviluppo grafico dell'interfaccia e il test dell'applicazione comprensivo anche della possibilità di emulare un dispositivo. Il linguaggio di programmazione che si utilizza può essere scelto al momento della creazione del nuovo file tra Java e Kotlin; in particolare, Java è quello più datato mentre Kotlin è più recente e viene spesso utilizzato dai programmatori perché più compatto e meno laborioso [7].

È stato scelto di utilizzare il linguaggio Java perché, seppur più datato, risulta di più facile comprensione anche per chi si è avvicina per la prima volta a tale linguaggio. La simulazione dell'applicazione è avvenuta su un Tablet Android collegato via cavo al computer su cui, il software di programmazione, aveva l'autorizzazione ad installare e attivare l'applicativo. Ad ogni test, il codice dell'applicazione veniva aggiornato e salvato nel dispositivo, di conseguenza, anche a tablet scollegato, è stato possibile utilizzare l'applicazione [8].

Durante l'attività di tirocinio è stato valutato anche l'utilizzo di altre possibili alternative di software come, ad esempio, l'estensione per app mobile di Simulink, ma è risultato troppo poco flessibile e non adatta allo scopo.

2.1 Configurazione iniziale

Alla prima apertura del programma, viene richiesta la creazione di un nuovo progetto tramite un'apposita schermata all'interno della quale si può scegliere il tipo di dispositivo per cui si vuole sviluppare l'applicazione (smartphone e tablet, Wear OS per orologi, Television o Automotive) e la base dell'interfaccia da cui partire.

Successivamente è possibile configurare il nome del progetto, il linguaggio di programmazione (Java o Kotlin), la destinazione in cui verrà salvato il progetto e anche l'SDK ossia il parametro che permette di definire la compatibilità con i vari dispositivi in base alla versione di Android minima sulla quale si vuole eseguire l'applicazione. In particolare, per quest'ultimo, più bassa sarà la versione minima utilizzabile e maggiore sarà il numero di dispositivi con cui sarà compatibile l'applicativo. Dopo questa fase di "configurazione iniziale", il programma procede con la creazione degli "strumenti" necessari al funzionamento dell'applicazione e, una volta terminato tale processo, è possibile dare inizio all'effettiva programmazione. [9]

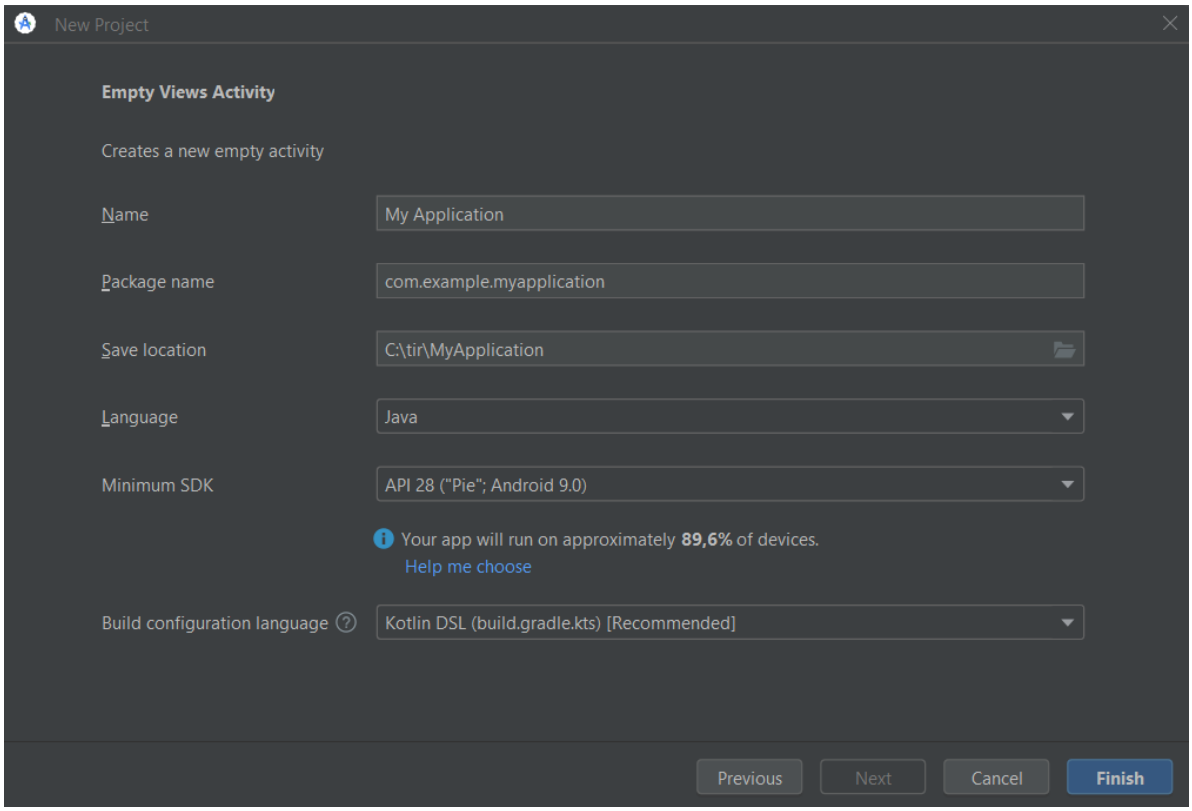

Figura 1. Pagina di configurazione iniziale dell'applicazione

2.2 Sviluppo Grafico (file.xml)

La programmazione avviene principalmente in due parti differenti che comunicano nel modo opportuno, ovvero la parte di sviluppo grafico e quella di sviluppo del codice. Quella di sviluppo grafico è effettuata nel file ActivityMain.xml, presente vicino al bordo dello schermo in alto a sinistra in Project e seguendo questo percorso:

nome_file -> app -> src -> main -> res -> layout.

In questa fase si utilizza un'altra delle peculiarità di Android Studio che consente di realizzare la programmazione grafica in diversi modi.

È possibile operare esclusivamente tramite codice, modalità Code, oppure utilizzando dei blocchi predefiniti, selezionandoli da una lista e posizionandoli sull'interfaccia grafica. Il programma, in questa modalità definita Design, genera automaticamente il codice corrispondente.

Vi è infine una modalità ibrida, denominata Split, in cui lo schermo è diviso in due sezioni: una dedicata al codice e l'altra alla visualizzazione grafica dell'applicazione.

Nella figura 2 è stato evidenziato in rosso il selettore per le 3 modalità di programmazione sopra descritte (modalità Code, Design, Split).

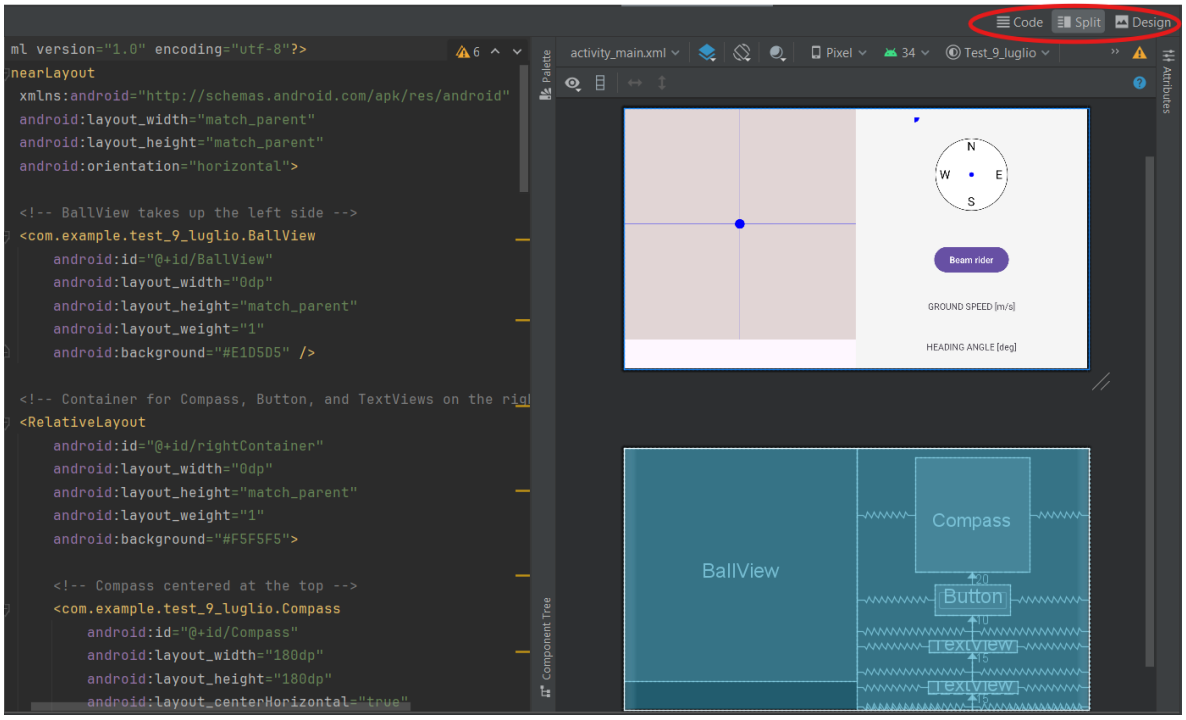

Figura 2. Schermata per lo sviluppo grafico dell'applicazione in modalità Split (file .xml)

Focalizzandosi sulla parte di codice nel file.xml, si può notare come ogni elemento presente nella schermata possieda un suo blocco specifico di codice in cui è possibile settare la posizione all'interno della visualizzazione, la dimensione, lo sfondo e moltissimi altri parametri relativi allo sviluppo grafico. È fondamentale

assegnare un id ad ogni blocco; tale denominazione permetterà di dialogare con la parte di codice che verrà analizzata successivamente [8].

2.3 Sviluppo del codice (file.java)

Il codice effettivo dell'applicazione si trova nel file MainActivity.Java seguendo il percorso: *Project -> nome_file -> app -> src -> main -> java ->com.example.nomefile*.

Prima di procedere con la descrizione dei file.Java, si ritiene doveroso fare una precisazione sul linguaggio di programmazione utilizzato.

Il linguaggio Java si basa sulle classi, ovvero delle entità che al loro interno contengono sia i dati sia il codice che manipolerà quelle informazioni, specificandone anche il livello di protezione (pubblico o privato) che è fondamentale per autorizzare le altre classi all'accesso alla classe stessa. In particolare, però, all'interno delle classi, non si troveranno mai delle istruzioni come nei linguaggi di programmazione usuali ma sono presenti dei metodi che a loro volta contengono le diverse istruzioni. Si ritiene importante sottolineare che void esegue soltanto delle azioni ma non restituisce alcun tipo di dato.

MainActivity è la classe in cui sono raccolti i metodi principali del codice e da dove vengono richiamati i metodi delle altre classi. Il file MainActivity.java è formato, in primo luogo, da una lista di pacchetti che devono essere importati in base agli strumenti che vengono utilizzati nel codice. Questa attività viene eseguita in automatico ogni volta che si richiama un certo comando. Successivamente si aprirà la public class, con le dovute estensioni, che a sua volta conterrà una serie di metodi che verranno eseguiti solamente in particolari occasioni come, per esempio, in fase di creazione dell'applicazione, alla modifica di un parametro che si sta monitorando tramite i sensori interni oppure tramite una chiamata effettuata da un'altra parte di codice.

Prima dei metodi è presente una serie di definizioni degli oggetti utili durante l'esecuzione della classe. Si ritiene opportuno sottolineare che gli oggetti definiti fuori dai metodi, possono essere utilizzati da tutti i metodi presenti in quella classe mentre, se si inizializza un oggetto dentro uno specifico metodo, esso è accessibile

solamente da quello specifico metodo. Le variabili accessibili da tutte le classi vengono evidenziate dal software con il colore viola mentre quelle locali inizializzate dentro i metodi, sono di colore bianco.

Durante la fase di programmazione è possibile creare anche altre classi che a loro volta conterranno altri metodi e che potranno essere richiamati all'interno del MainActivity tramite il comando: *classe.metodo*.

L'ambiente di sviluppo del codice è molto intuitivo, segnala gli errori in tempo reale durante la sua stesura e propone anche alcune soluzioni per correggere eventuali problemi rilevati. Gli errori e i warning (consigli) vengono segnalati rispettivamente con una linea rossa orizzontale e con una linea gialla orizzontale all'interno di un rettangolo posto sulla destra dell'ambiente di sviluppo del codice. Gli errori impediscono la compilazione e il test dell'applicazione mentre i warning, no.

Figura 3. Alla riga 195 è stato rimosso il punto e virgola finale: in rosso i dove punti viene segnalato l'errore (e il nome del file diventa sottolineato)

3. ALGORITMO DI GUIDA

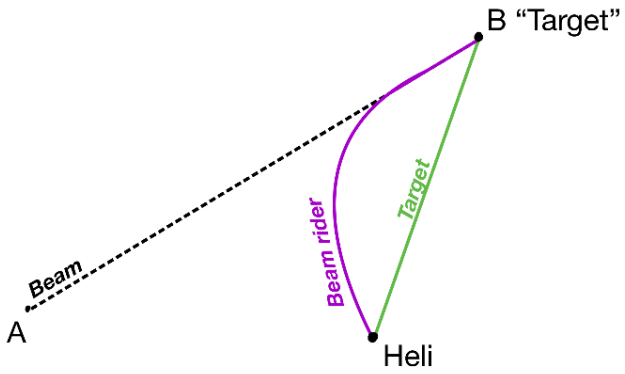

Figura 4. in nero la traiettoria ideale, in verde quella diretta al target e, in viola, la traiettoria di Beam Rider

Il nucleo centrale dell'applicazione, ovvero colui che governa il movimento della pallina, è l'algoritmo di guida, ossia un insieme di istruzioni di tipo trigonometrico che permettono di trovare l'angolo di rotta desiderato (χ),per raggiungere il target "B".

Si ricorda che l'angolo di rotta è l'angolo definito tra la direzione del vettore velocità e il vettore in direzione Nord sul piano orizzontale [2]. Per raggiungere il target è possibile scegliere tra due differenti strategie:

1) trovare l'angolo di rotta tale che ci si diriga immediatamente in direzione di B a prescindere della traiettoria ideale (Target);
2) calcolare l'angolo di rotta che permette di descrivere una traiettoria curvilinea che si immette nella traiettoria desiderata ovvero quella che va dal punto iniziale (A) a quello finale (B).

Quest'ultimo metodo, denominato Beam Rider, è ampiamente diffuso nell'ambito missilistico in quanto, tramite il fascio creato da un puntatore laser orientato verso l'obiettivo, il missile si immetterà automaticamente nella traiettoria definita dal fascio e la manterrà fino al raggiungimento del target [10]. Si è pensato quindi di applicare questa metodologia anche nel campo dell'aviazione al fine di far inserire e mantenere il velivolo nella rotta corretta definita tra il punto di partenza e quello di arrivo. L'intero sistema è stato studiato appositamente per fornire anche le correzioni agli eventuali sbandamenti durante l'intera missione.

Osservando i vettori velocità di entrambe le soluzioni si ha, nel primo caso, che il vettore velocità rimane in direzione costante verso l'obiettivo mentre, nel secondo caso, il vettore velocità è il risultato della somma tra un vettore posto in direzione della traiettoria e uno posto in direzione del target così che, durante l'avanzamento, ci si avvicini contemporaneamente al target e al tragitto ottimale. Ovviamente, nel secondo caso, il vettore velocità varia progressivamente al mutare della posizione.

Nell'applicazione per smartphone oggetto di questa relazione, è possibile selezionare una delle due strategie tramite un pulsante apposito che, se attivato, permette di utilizzare l'algoritmo del Beam Rider.

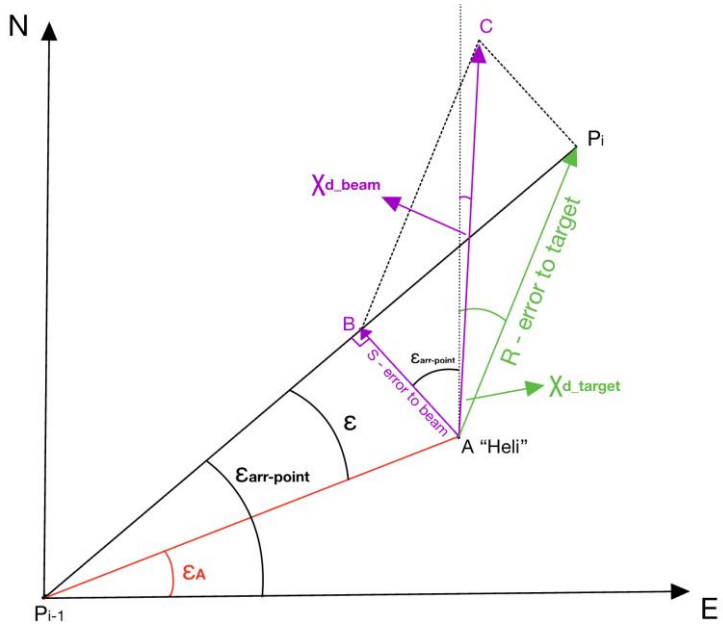

Figura 5. Schematizzazione dell'algoritmo del beam rider

Passando ora all'analisi dell'algoritmo vero e proprio, risulta di estrema importanza lo schema riportato in Figura 5 [11]. Si precisa che l'algoritmo sta lavorando in assi NED [12], di conseguenza è necessario fare molta attenzione in quanto, sull'asse delle ordinate, è presente la prima delle tre componenti del vettore delle coordinate.

Per il corretto utilizzo dell'algoritmo, è necessario convertire nello stesso sistema NED il vettore della posizione istantanea, quello del punto iniziale e quello del punto finale.

Si definiscono quindi i seguenti vettori già convertiti in assi NED:

- Arr_point [3] ->punto di arrivo;
- Init_point [3] -> punto iniziale;
- NED[3] ->posizione istantanea del velivolo.

Successivamente si individuano gli angoli ε (epsilon) tramite l'arcotangente a quattro quadranti facendo attenzione al sistema NED:

$$\varepsilon_{Arr_{point}} = Arctg2\ (Arr\_point[1] - Init\_point[1],\ Arr\_point[2] - Init\_point[2]) \quad (1)$$

$$\varepsilon_A = Arctg2\ (NED[1] - Init\_point[1]\ ,\quad NED[2] - Init\_point[2]\ ) \quad (2)$$

$$\varepsilon = \varepsilon_{Arr_{point}} - \varepsilon_A \quad (3)$$

Una volta determinato ε *(3)* si può individuare facilmente S *(4)* ossia la distanza del velivolo dal beam, solamente dopo aver calcolato il modulo del vettore che congiunge il punto di partenza con la posizione istantanea del velivolo:

$$S = sqrt(\left(NED[1] - Init_{point[1]}\right)^2 + \left(NED[2] - Init_{point[2]}\right)^2) * sin(\varepsilon); \quad (4)$$

Allo stesso modo di S, è possibile individuare anche il vettore R (*5)* come differenza tra la posizione del target e quella del velivolo e successivamente si calcola anche il modulo *(6)*

R_NED= {

Arr_point[1] - NED[1],

Arr_point[2] - NED[2] } *(5)*

R =sqrt(R_NED[1] ^2 + R_NED[2] ^2); *(6)*

Con R è possibile calcolare χd_target *(7)* ovvero l'angolo χ da utilizzare per dirigersi direttamente al target senza immettersi nel Beam:

$$\chi d_{target} = arctg2\ (R\_NED[2],\ R\_NED[1])\ ; \quad (7)$$

Tramite χd_{target} si individua il vettore AC *(8)* che è dato dalla somma vettoriale tra S e R ed è quello che permette di individuare l'angolo χ da utilizzare se si vuole utilizzare la metodologia Beam Rider *(9)*.

$$C\_NED = \{ \quad R * cos(\chi d_{target}) + S * cos(\varepsilon_{Arr_{point}}), \quad R * sin(\chi d_{target}) - S * sin(\varepsilon_{Arr_{point}}) \} \qquad (8)$$

$$\chi d_{beam} = arctg2(C\_NED[2],\ C\_NED[1]) \qquad (9)$$

All'interno dell'applicazione, questo algoritmo viene eseguito ad ogni cambio di posizione del velivolo, perciò, il vettore C e quindi l'angolo χd_{beam}, variano ad ogni istante. Si osserva che, inizialmente, il vettore C tenderà ad orientarsi verso la traiettoria ottimale; tuttavia, man mano che ci si avvicina ad essa, il vettore C si allineerà progressivamente verso il target, fino a raggiungere il momento in cui indicherà esattamente il target, quando questo coinciderà con il fascio (beam).

Successivamente, una volta ottenuti gli angoli (χd_{target} e χd_{beam}) e avere scelto quello di riferimento, esso verrà relazionato con l'angolo di rotta effettivo (calcolato dal GPS) tramite una differenza, il cui risultato verrà assegnato alla variabile error *(10)* che è quella rappresentata sulle ascisse del grafico con la pallina. Sulle ordinate si troverà invece una differenza tra la quota del target e quella attuale.

$error = \chi_{ref} - \chi$ con χ_{ref} che è uno tra χd_{target} e χd_{beam} *(10)*

Come già detto, la pallina rappresenta la rotta ottimale mentre il centro rappresenta il velivolo; di conseguenza, per minimizzare l'errore tra la rotta del velivolo e quella ottimale, bisognerà compiere ogni manovra nella direzione in cui si trova la pallina cioè, ipotizzando di avere la pallina nel secondo quadrante, sarà necessario virare a sinistra e cabrare (logicamente, sarà il contrario nel caso in cui fossimo nel quarto quadrante).

4. ANALISI DEL CODICE

Al fine di comprendere pienamente il funzionamento dell'applicazione, si procede con l'analisi di ogni sezione del codice.

Come già detto in precedenza, all'interno di Android Studio si lavora per classi e, per svolgere al meglio il lavoro assegnato, ne sono state utilizzate di ulteriori rispetto a quella principale di MainActivity.Java. Ogni classe aggiuntiva verrà analizzata in un paragrafo dedicato di questo capitolo.

Si ritiene utile precisare che, in questo linguaggio di programmazione, gli indici relativi alla posizione dei vari elementi nei vettori o nelle matrici partono da 0 cioè la prima posizione di un array viene indicata dall'indice 0. Per quanto riguarda la creazione, invece, è necessario indicare il numero effettivo di elementi che si vuole inserire nel vettore; di conseguenza, se si volesse utilizzare un vettore di 3 componenti, alla definizione occorrere impiegare il numero 3 mentre l'indice massimo utilizzabile sarà 2 (che si riferisce alla terza cella) [7].

4.1 MainActivity.Java

La classe MainActivity.Java è la parte più importante del codice in quanto coordina ogni processo che avviene all'interno dell'applicazione.

```
public class MainActivity extends AppCompatActivity implements LocationListener {
  //definizioni per coordinate LLA
  private static final int PERMISSION_REQUEST_CODE = 1;
  private static final String TAG = "MainActivity";
  private final String providerId = LocationManager.GPS_PROVIDER;
  private Location location;
  private LocationManager locationManager = null;
  private static final int MIN_DIST = 75 * 10 ^ (-2)
private static final int MIN_PERIOD =200 ;
```

Il file relativo alla classe mainActivity.Java si apre con una serie di importazioni di alcuni pacchetti che vengono eseguite in automatico in base agli strumenti utilizzati e che, in questa sede, non vengono riportate.

Successivamente, si inizializza la classe vera e propria, definita pubblica, estesa al pacchetto AppCompactActivity e si importa il LocationListener.

L'AppCompactActivity viene utilizzata come base per creare attività compatibili con le versioni più recenti di Android mentre il LocationListener è necessario per ricevere gli aggiornamenti relativi alla posizione del dispositivo [13].

I comandi preceduti da doppio Slash sono "commenti" e servono solamente a rendere più chiaro il codice. In questo primo blocco si trovano le definizioni delle variabili necessarie per ottenere le coordinate GPS del dispositivo. Tali variabili vengono definite tutte private in quanto non si vuole abilitare l'accesso da altre classi, se non da MainActivity.Java.

Nella tabella seguente sono elencate e descritte le differenti tipologie di variabili:

CATEGORIA	PAROLA CHIAVE O MODIFICATORE	DESCRIZIONE
Modificatori di accesso	'public'	Indica che la classe, metodo o variabile è accessibile da qualsiasi altra classe.
	'private'	Indica che la classe, metodo o variabile è accessibile solo all'interno della classe in cui è definita.
	'protected'	Indica che la classe, metodo o variabile è accessibile all'interno dello stesso pacchetto o dalle sottoclassi.
Modificatori di classe	'final'	Impedisce la modifica di una variabile (il suo valore non può essere cambiato una volta assegnato). Se usato con una classe, impedisce l'ereditarietà; se usato con un metodo, impedisce l'override.
	'static'	Indica che la variabile o il metodo appartiene alla classe piuttosto che a una specifica istanza di quella classe
Tipi di dati primitivi	'int'	Tipo di dato per numeri interi.
	'float'	Tipo di dato per numeri in virgola mobile a precisione singola.
	'double'	Tipo di dato per numeri in virgola mobile a doppia precisione.
	'boolean'	Tipo di dato per valori booleani (true o false).

Tabella 1. Tipologie più comuni di variabili

Procedendo all'analisi riga per riga dell'ultima parte di codice presentata si precisa quanto segue:

- PERMISSION_REQUEST_CODE: codice di richiesta per gestire i permessi. Questa costante rappresenta un codice di richiesta che viene utilizzato quando si chiede una specifica autorizzazione;
- TAG: identificatore per il logcat (strumento per il debug) che permette di segnalare che ogni messaggio ha origine in una determinata classe;
- ProviderId: identificatore del provider ovvero si segnala che il provider da cui si preleva l'informazione è il GPS;
- location: variabile che memorizza la posizione dell'utente;
- locationManager: gestore della posizione;
- MIN_DIST: distanza minima per aggiornare la posizione in metri;
- MIN_PERIOD: periodo minimo tra gli aggiornamenti della posizione in millisecondi.

MIN_DIST è stato impostato a 0.75 mentre MIN_PERIOD a 200 in modo tale che l'aggiornamento della posizione venga effettuato con un periodo di 200ms ovvero a 5Hz.

```
//Definizioni per TEXT VIEW
  private TextView V_attuale_TextView;
  private TextView Ki_TextView;
  private TextView Distance_TextView;
 private TextView delta_speed_TextView;
 private TextView state_TextView;
 private TextView delta_speed_TextView;
 private TextView altitude_TextView;
 private BallView ballView;
 private Compass compass;
 private Button button;
```

In questo blocco sono contenute tutte le definizioni necessarie per le textview ovvero quei blocchi che permettono la visualizzazione testuale dei dati elaborati nell'interfaccia grafica. Nelle prime righe, dopo la definizione come privati, si trova la definizione del tipo di dato in Android che rappresenta un widget dell'interfaccia utente.

In particolare, TextView è necessario per la visualizzazione del testo sullo schermo mentre BallView è utilizzato per la visualizzazione della pallina, Compass per la bussola e Button per il tasto.

In seguito, dopo il widget si trova la variabile effettiva che si vuole inizializzare all'interno di quel widget.

```
//Definizioni per il Beam rider
private double GS;
private boolean Beam_rider_choice = false;
private double ki_ref;
private double[] ARR_point_LLA = new double[4];
private double[] Init_point_LLA = new double[3];
private  double[] Init_point = new double[3];
private  double[] ARR_point = new double[3];
private double ki; //angolo di rotta
double[] Beam_rider_output = new double[3];
```

In questa fase sono riportate tutte le definizioni delle variabili necessarie per utilizzare i risultati provenienti dall'algoritmo del Beam Rider. Sono state definite tutte variabili double ma anche dei vettori a quattro o a tre dimensioni (sempre double) per inserire i dati dei punti di arrivo e di partenza (quelli che terminano con LLA). Si trovano inoltre due vettori aggiuntivi necessari per memorizzare le coordinate dei punti di arrivo e di partenza in coordinate NED. Ki è la variabile che memorizza l'angolo di rotta.

```
// Definizioni per la conversione delle coordinate
private double[] LLA = new double[3];
private double[] ECF = new double[3];
private double[] LLA_zero = new double[3];
private double[] ECEF_zero = new double[3];
private double[] NED = new double[3];
```

In queste righe si trova la definizione di tutti i vettori necessari alla conversione delle coordinate GPS dal sistema LLA (longitudine, latitudine, quota) a NED, passando necessariamente per il sistema ECEF. Per la conversione in assi NED è necessario individuare un'origine del sistema di riferimento che viene salvata all'ottava iterazione tramite l'inserimento nei vettori corrispondenti.

```
// Definizioni per il filtro
private int counter_filtro=0;
```

```
private double[] [] filter_matrix= new double[3][2];
private double[]  ki_matrix= new double[3];
private double delta_speed;
private int counter = 0;
private FusedLocationProviderClient fusedLocationClient;
```

Questa sezione contiene le definizioni di tutte le variabili adibite al filtraggio dei dati in ingresso che si rende necessario al fine di stabilizzare i dati stessi.

Oltre agli strumenti di filtraggio si definiscono le variabili relative all'errore di velocità e al contatore utilizzato per il conteggio delle iterazioni.

La variabile fusedLocationClient è un'istanza della classe FusedLocationProviderClient, che rappresenta il client utilizzato per accedere ai servizi di geolocalizzazione. Questo client permette di ottenere la posizione del dispositivo in modo più efficiente rispetto alle tradizionali API di Android, poiché combina più fonti di localizzazione (GPS, Wi-Fi, reti mobili, ecc.) per migliorare la precisione e risparmiare energia.

```
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  locationManager = (LocationManager)getSystemService(LOCATION_SERVICE);
  fusedLocationClient = LocationServices.getFusedLocationProviderClient(this);
  setContentView(R.layout.activity_main);
  V_attuale_TextView= findViewById(R.id.V_attuale_TextView);
  Ki_TextView = findViewById(R.id.Ki_TextView);
  Distance_TextView = findViewById(R.id.Distance_TextView);
  delta_speed_TextView= findViewById(R.id.delta_speed_TextView);
  state_TextView= findViewById(R.id.state_TextView);
  delta_altitude_to_target_TextView= findViewById(R.id.delta_altitude_to_target_TextView);
  altitude_TextView = findViewById(R.id.altitude_TextView);
  Button button = (Button) findViewById(R.id.Beam_rider_button);
  ballView = findViewById(R.id.BallView);
  compass = findViewById(R.id.Compass);
  state_TextView.setText("WAIT");
  state_TextView.setTextColor(0xFFFF0000);
  button.setBackgroundColor(getResources().getColor(R.color.grey));
 button.setOnClickListener(new View.OnClickListener() {

    public void onClick(View view) {
      if (Beam_rider_choice == false) {
        Beam_rider_choice = true;
        button.setBackgroundColor(getResources().getColor(R.color.green));
      } else {
```

```
    Beam_rider_choice = false;
    button.setBackgroundColor(getResources().getColor(R.color.grey));
  } } };
```

Si procede ora ad analizzare i vari metodi presenti in MainActivity.Java.

Il primo blocco è onCreate ovvero un metodo che viene eseguito alla creazione dell'applicazione e all'interno del quale si configurano gli strumenti necessari. In ingresso a questo metodo viene fornito il Bundle savedInstanceState, necessario al recupero dello stato in cui si trova l'activity antecedente ad un'interruzione. Infatti, anche solo la rotazione della vista (da orizzontale a verticale o viceversa) crea una ricostruzione della stessa pertanto, senza questo Bundle, le variabili e i codici andrebbero persi nella fase di ricostruzione.

Nei comandi successivi si trova la definizione del LocationManager che coordina l'intera parte dei servizi di localizzazione del dispositivo e, in questa sezione, vengono attivati tali servizi. Alla riga successiva si trova inoltre un comando che permette l'attivazione del servizio di localizzazione sia tramite GPS che tramite Wi-Fi (o rete mobile) così da assicurare una maggiore velocità e precisione nella ricerca delle coordinate.

setContentView(R.layout.activity_main) è un comando fondamentale in quanto stabilisce il collegamento tra il codice Java e il file XML che gestisce il layout dell'applicazione.

Da questo punto in avanti, viene introdotto un insieme di comandi che associano, ogni variabile ad un ID precedentemente inizializzato nel file.xml. In sostanza, per ogni informazione da visualizzare nella schermata dell'applicazione, viene definita una corrispondente variabile.

Successivamente, con il comando *variabile_in_java*=findViewById(R.id.*id_nell'XML*), la variabile a sinistra dell'uguale viene collegata a quella specificata all'interno delle parentesi, corrispondente all'ID presente nel file.xml. Per semplicità è stato usato lo stesso nome per le variabili e per gli ID ma ciò non è necessario. Dopo queste definizioni per le parti testuali, si fornisce il comando di associazione tra file java e l'xml per il bottone, per la ballview e per la bussola.

Il widget state_textview è necessario al fine di indicare all'utente il termine della fase di inizializzazione del sistema. Più precisamente, all'avvio del codice, comparirà nell'apposito widget la dicitura rossa "WAIT" mentre, quando sarà possibile iniziare con gli aggiornamenti effettivi della posizione e di tutte le variabili necessarie, allora comparirà "READY" in verde. Si precisa che per l'impostazione di una stringa all'interno di un certo widget è necessario utilizzare il comando *variabile*.setText("*testo da visualizzare*"+*eventuale variabile*).

Il blocco di codice si chiude con la parte relativa all'utilizzo del bottone in quanto si imposta il listener (cioè ci si pone in ascolto delle sue variazioni di stato) e in seguito, si stabiliscono una serie di azioni da compiere nel momento in cui viene premuto (onClick). Si verifica il relativo stato e conseguentemente si aggiorna la variabile "beam_rider_choice" necessaria per la scelta dell'angolo di riferimento tra i due possibili. È presente, inoltre, un'istruzione che imposta il colore dello sfondo del bottone in base alla scelta effettuata ovvero, se si attiva il beam rider, il tasto presenterà uno sfondo di colore verde.

```
@Override
protected void onResume() { //fa partire l'aggiornamento della posizione
  super.onResume();
  if (!locationManager.isProviderEnabled(providerId)) {
    Log.w(TAG, "onResume: GPS provider not enabled");
    Intent gpsOptionsIntent = new Intent(Settings.ACTION_LOCATION_SOURCE_SETTINGS); //se il
GPS vede che la posizione è disattivata, noi non possiamo riattivarla ma apriamo le impostazioni del
dispositivo così che l'utente possa attivarla
    startActivity(gpsOptionsIntent);
  } else {
    if (ContextCompat.checkSelfPermission(this, Manifest.permission.ACCESS_FINE_LOCATION) !=
PackageManager.PERMISSION_GRANTED
        && ContextCompat.checkSelfPermission(this,
Manifest.permission.ACCESS_COARSE_LOCATION) != PackageManager.PERMISSION_GRANTED) {
      Log.d(TAG, "onResume: Requesting location permissions");
      ActivityCompat.requestPermissions(this, new
String[]{Manifest.permission.ACCESS_FINE_LOCATION,
Manifest.permission.ACCESS_COARSE_LOCATION}, PERMISSION_REQUEST_CODE);
    } else {
      Log.d(TAG, "onResume: Permissions granted, starting location updates");
      startLocationUpdates();

}  } }
```

In questo metodo, che viene attivato nella fase di ripresa dell'aggiornamento della posizione, vengono effettuati tutti i controlli relativi alla corretta attivazione del servizio di localizzazione. Infatti, nella prima parte, nel caso in cui il servizio risulti disattivato dall'utente, si apre direttamente la pagina delle impostazioni del dispositivo così che l'utilizzatore possa riattivarlo. In aggiunta, nelle fasi successive, si procede al controllo dell'autorizzazione che deve possedere l'applicazione per accedere alla posizione sia di tipo COARSE che quella FINE (più precisa). Successivamente alla verifica di tali autorizzazioni, è possibile iniziare con l'aggiornamento della posizione.

Si ricorda che i comandi Log servono al programmatore per comprendere quello che il codice sta eseguendo in ogni istante tramite l'interfaccia Logcat.

```
@Override
public void onRequestPermissionsResult(int requestCode, @NonNull String[] permissions,
@NonNull int[] grantResults) {
  super.onRequestPermissionsResult(requestCode, permissions, grantResults);
  if (requestCode == PERMISSION_REQUEST_CODE) {
    if (grantResults.length > 0 && grantResults[0] == PackageManager.PERMISSION_GRANTED) {
      Log.d(TAG, "onRequestPermissionsResult: Permissions granted, starting location updates");
      startLocationUpdates();
    } else {
      Log.w(TAG, "onRequestPermissionsResult: Permissions denied");
    } } }
```

onRequestPermissionsResult è un metodo che viene richiamato dopo che sono state concesse o revocate delle autorizzazioni ed è necessario per verificare tali autorizzazioni. Nel caso fosse tutto verificato, si attiva l'aggiornamento della posizione.

```
private void startLocationUpdates() {
  if (ActivityCompat.checkSelfPermission(this, Manifest.permission.ACCESS_FINE_LOCATION) ==
PackageManager.PERMISSION_GRANTED
      && ActivityCompat.checkSelfPermission(this,
Manifest.permission.ACCESS_COARSE_LOCATION) == PackageManager.PERMISSION_GRANTED) {
    locationManager.requestLocationUpdates(providerId, MIN_PERIOD, MIN_DIST, this);
    Log.d(TAG, "LocationUpdates @start_location_updates");

    Location lastknownLocation = locationManager.getLastKnownLocation(providerId);
    if (lastknownLocation != null) {
      updateGUI(lastknownLocation);
```

```
        Log.d(TAG, "Last known location");
    } }}
```

startLocationUpdates() è il metodo, già richiamato in precedenza, che permette di attivare l'aggiornamento della posizione. Tale aggiornamento viene effettuato soltanto se è presente l'accesso alla posizione FINE e COARSE. In questo caso, tramite il comando *locationManager.requestLocationUpdates(providerId, MIN_PERIOD, MIN_DIST, this)* viene aggiornata la posizione, previo inserimento di ulteriori parametri come il minimo periodo di aggiornamento, la minima distanza di aggiornamento e la classe in cui avviene il rilevamento.

L'ultima parte di codice salva, come prima posizione, l'ultimo punto noto all'interno dell'applicativo così da poterlo utilizzare nel momento dell'avvio. Soltanto se l'ultima posizione conosciuta è diversa da 0, allora essa viene importata nella funzione vera e propria che aggiorna l'interfaccia grafica.

```
@Override
protected void onPause() //interrompe l'aggiornamento della posizione
{ super.onPause();
   if (locationManager != null) {
      locationManager.removeUpdates(this);}}
```

onPause() serve per interrompere l'aggiornamento della posizione una volta che l'applicativo viene messo in pausa.

```
@Override
public void onLocationChanged(Location location) //richiamiamo un metodo privato scritto sotto
quando la posizione cambia
{
   Log.d("LocationUpdate", "Location changed: " + location.toString());
   updateGUI(location);
   }
```

Questo metodo privato viene richiamato ogni qual volta si rileva una variazione della posizione ovvero un cambiamento della stessa che sia al di fuori delle limitazioni di tempo e di distanza impostati nelle variabili MIN_DIST e MIN_PERIOD. Ogni modifica della posizione viene salvata all'interno della variabile "location" che a sua volta, ad ogni variazione, richiama updateGUI che si occupa di aggiornare l'interfaccia grafica.

```
private void updateGUI(Location location) //qui preleviamo latitudine e longitudine e le inseriamo
nelle variabili
{ if (counter==8) {
```

```
    LLA_zero[0] = LLA[0];
    LLA_zero[1] = LLA[1];
    LLA_zero[2] = LLA[2];
    ECEF_zero[0] = ECEF[0];
    ECEF_zero[1] = ECEF[1];
    ECEF_zero[2] = ECEF[2];
    Log.d(TAG, "zero_location updated");


// inserimento e conversione delle coordinate del punto di arrivo
    ARR_point_LLA[0] =44.428515;
    ARR_point_LLA[1] =12.201029;
    ARR_point_LLA[2] =0; //altitudine del punto di arrivo
    ARR_point_LLA[3]=1; //velocità di arrivo sul target a 10m/s
    double [] Arr_point_ecef =Converter.lla_to_ecef(ARR_point_LLA);
    ARR_point=Converter.ecef_to_NED(Arr_point_ecef, ECEF_zero, LLA_zero); //ora abbiamo il punto di
arrivo in coordinate NED
    Log.d(TAG, "arr_point ned: "+ARR_point[0] +" lat; " +ARR_point[1]+"long; "+ARR_point[2]+"alt");


// inserimento e conversione delle coordinate del punto di partenza
    Init_point_LLA[0] =44.428314;
    Init_point_LLA[1] =12.201320;
    double [] Init_point_ecef =Converter.lla_to_ecef(Init_point_LLA);
    Init_point=Converter.ecef_to_NED(Init_point_ecef, ECEF_zero, LLA_zero); //punto di partenza nel
sistema NED
    Log.d(TAG, "Init_point ned: "+Init_point[0] +" lat; " +Init_point[1]+"long; "+Init_point[2]+"alt");


    state_TextView.setText("READY");
    state_TextView.setTextColor(0xFF00FF00);

}
```

Il metodo updateGUI viene attivato ad ogni cambiamento della posizione del dispositivo ed utilizza come ingresso la posizione precedentemente salvata in Location location. Il primo blocco che si trova in questo metodo viene eseguito solamente se la variabile “counter”, che viene incrementata ad ogni variazione della posizione e quindi ad ogni chiamata del metodo updateGUI, risulta uguale a 8. Quando questa condizione risulta verificata, vengono memorizzate in vettori appositi le coordinate geografiche di quell’istante sia nel sistema LLA che nel sistema ECEF in quanto necessarie in entrambi i formati. La conversione nei diversi sistemi di riferimento verrà analizzata in seguito. Purtroppo, in questo linguaggio di programmazione, non è possibile eguagliare direttamente due vettori quando si vuole

copiare il contenuto fra essi, quindi, sarà necessario svolgerlo elemento per elemento. Successivamente alla registrazione della posizione di riferimento, si convertono anche le posizioni del punto di arrivo e di partenza in assi NED (passando per quello ECEF) e si salvano nel vettore corrispondente. Al momento questi punti vengono forniti via software alla creazione dell'applicazione tramite l'inserimento delle coordinate nel sistema LLA. Si segnala che le coordinate nel sistema LLA devono essere inserite in formato decimale e non nel formato standard che prevede l'utilizzo di gradi, primi e secondi.

Si ritiene doveroso sottolineare che il vettore tramite il quale si inseriscono le coordinate del punto di arrivo (Arr_Point_LLA) è formato da 4 elementi in quanto, oltre a contenere gli usuali 3 parametri per la georeferenziazione, si inserisce anche un quarto parametro che rappresenta la velocità con cui si vuole raggiungere il punto.

La conclusione di questo blocco decisionale implica la fine della fase di inizializzazione e l'inizio della fase di utilizzo dell'applicazione. Tale cambio di circostanza viene comunicata all'utente tramite il cambio di dicitura nel widget dello stato da "WAIT" a "READY" e il cambio di colore del carattere da rosso a verde.

```
counter = counter + 1;
//----------------------position filter-------------------
filter_matrix[counter_filtro][0]= location.getLatitude();
filter_matrix[counter_filtro][1]= location.getLongitude();
ki_matrix[counter_filtro] = location.getBearing();
counter_filtro=counter_filtro+1;
Log.d(TAG, "counter_filtro changed: " +counter_filtro);

if (counter_filtro==3){counter_filtro=0;}

LLA[0]=(filter_matrix[0][0]+filter_matrix[1][0]+filter_matrix[2][0])/3;
LLA[1]=(filter_matrix[0][1]+filter_matrix[1][1]+filter_matrix[2][1])/3;
ki=(ki_matrix[0]+ki_matrix[1]+ki_matrix[2])/3;

LLA[2] = location.getAltitude();
altitude_TextView.setText("" + Math.round(LLA[2]));
```

In seguito al blocco decisionale, che in base allo stato del contatore può essere eseguito oppure no, inizia la parte di codice che viene sicuramente eseguita ad ogni chiamata del metodo "updateGUI". Oltre al contatore "counter" che viene aggiornato ad ogni chiamata del metodo in esame, si trovano una serie di comandi che

implementano un filtro che stabilizza il dato relativo alla posizione e all'angolo di rotta (ki).

Per il filtro della posizione, è necessario inizializzare una matrice di tre righe e due colonne chiamata filter_matrix. In tale matrice, ad ogni interazione, si salva il valore istantaneo della latitudine e della longitudine in una delle tre righe definite dall'apposito contatore "counter_filtro". I valori istantanei della posizione si ottengono rispettivamente con i comandi location.getLatitude() e location.getLongitude(). Il contatore del filtro viene incrementato ad ogni iterazione ma viene resettato a 0 ogni qual volta che si raggiunge il valore 3. Tramite questa modalità si riesce ad eliminare dalla matrice il valore più datato mentre si conservano i valori istantanei e i due precedenti.

In seguito, ad ogni iterazione, il programma esegue la media dei valori di ogni colonna della matrice e la memorizza nel vettore LLA che è un input dell'algoritmo del beam rider. Con il comando location.getAltitude() il programma memorizza l'altitudine del dispositivo salvandola direttamente nel vettore LLA in quanto non necessita di stabilizzazione. In particolare, alla posizione LLA[0] è presente la latitudine, in LLA[1] la longitudine e in LLA[2] l'altitudine.

Un filtro identico a quello posizionale è utilizzato per normalizzare l'angolo di rotta istantaneo ottenuto tramite il comando location.getBearing(). L'unica differenza tra i due metodi è che, essendo necessario lavorare con un solo dato alla volta, si utilizza un vettore di tre dimensioni anziché una matrice.

```
ECEF = Converter.lla_to_ecef(LLA);
NED = Converter.ecef_to_NED(ECEF, ECEF_zero, LLA_zero);
Beam_rider_output = Beam_rider.error_calculation(NED, ARR_point, Init_point);
Distance_TextView.setText("" + String.format("%.2f", Beam_rider_output[2]));
```

In questo insieme di comandi si convertono le coordinate del vettore LLA appena ottenute nel sistema NED, passando necessariamente per il sistema ECEF. Le conversioni verranno analizzate in seguito nella classe "converter". Le coordinate istantanee appena convertite nel sistema NED e salvate nel vettore "NED", sono poi utilizzate come ingresso all'algoritmo del beam rider. Tale algoritmo necessita inoltre

del punto di arrivo e di partenza convertiti nello stesso sistema delle coordinate istantanee. Infine, si inserisce nel widget corrispondente il dato relativo alla distanza dal punto di arrivo, contenuto nella terza cella del vettore di output dell'algoritmo "Beam_rider_output".

```
//-------------------bearing acquisition-------------
 Ki_TextView.setText("" + Math.round(ki));
   Log.d(TAG, "Bearing  " + ki);
   compass.updatePosition(ki);
   ki = Converter.bearing_converter(ki);
// --------------------Ground speed acquisition
   GS = location.getSpeed();
   V_attuale_TextView.setText("" + Math.round(GS));

//-----------------------delta altitude ------------------------------
     delta_altitude_to_target_TextView.setText("" + Math.round((ARR_point_LLA[2] - LLA[2])));
```

In questo blocco di istruzioni si utilizza l'angolo di rotta precedentemente filtrato, come input per la visualizzazione grafica della "bussola" e anche come ingresso per la funzione di conversione del dato. Tale conversione si rende necessaria in quanto si desidera che l'angolo di rotta sia compreso tra 0 e 180° oppure tra 0 e -180° mentre, al momento, si trova all'interno del dominio [0°÷360°]. In seguito, si acquisisce la velocità al suolo salvandola nella variabile Ground Speed ("GS") e la si imposta nel widget corrispondente. Il comando relativo al delta_altitude è necessario per rappresentare il valore relativo alla differenza tra l'altitudine del target e quella attuale. Tale dato viene rappresentato graficamente anche nell'ordinata della BallView.

```
//-----------------delta speed-----------------------------------
   delta_speed = ARR_point_LLA[3] - GS;
   delta_speed_TextView.setTextSize(50);
   if (delta_speed >= 0) {
     delta_speed_TextView.setTextColor(0xFFFFA500); //arancio
     Log.d(TAG, "delta_speed: orange color updated " + delta_speed);
   } else {
     delta_speed_TextView.setTextColor(0xFFFF0000); //rosso
     Log.d(TAG, "delta_speed: red color updated " + delta_speed);
   }
   delta_speed_TextView.setText("" + Math.round(delta_speed));
```

In questa sezione di codice viene calcolata la differenza della velocità tra quella richiesta al passaggio sul target (contenuta nel vettore Arr_Point_LLA[3]) e quella

attuale (GS). In seguito a questo comando, si trovano una serie di istruzioni necessarie alla visualizzazione di tale dato in quanto si desidera che esso sia in una posizione privilegiata, di dimensioni maggiori rispetto agli altri e di un colore che cambi in base al segno dell'errore stesso. La dimensione del testo viene gestita tramite il comando *delta_speed_TextView.setTextSize(50)*.

In seguito, si trova un blocco decisionale "if" che permette di cambiare il colore in base al segno dell'errore di velocità; in particolare, se l'errore è maggiore di zero e quindi è necessario accelerare, il dato sarà arancione mentre, nel caso fosse necessario rallentare in quanto l'errore è negativo, allora il dato sarà visualizzato di colore rosso. Infine, si inserisce all'interno della relativa variabile "delta_speed_TextView" il dato calcolato in precedenza al quale sono state applicate le proprietà sopra descritte.

```
if (Beam_rider_choice == true) {
      ki_ref = Beam_rider_output[1];
      Log.d(TAG, "ki_ref beam rider  " + counter);
   } else {
      ki_ref = Beam_rider_output[0];
      Log.d(TAG, "ki_ref to target  " + counter);
   }
   double error = ki_ref – ki;
   ballView.updatePosition(error, ARR_point_LLA[2]-LLA[2]);
}
```

In quest'ultima sezione della classe MainActivity.Java viene prima stabilito l'angolo di riferimento da usare per il computo dell'errore, successivamente si calcola l'errore di rotta e lo si inserisce come ingresso alla ballview così che esso venga rappresentato. Più precisamente, all'interno del blocco "if", si verifica la variabile booleana di scelta che verrà aggiornata ad ogni pressione del tasto beamrider. Basandosi sul contenuto della variabile "Beam_rider_choice" si definisce il riferimento dell'angolo di rotta tra quello che individua direttamente il target (χd_{target}) oppure l'angolo di rotta che permette di inserirsi al meglio nella traiettoria ottimale (χd_{beam}). Questi due angoli, assieme alla distanza dal target, sono contenuti nel vettore di output della classe beamrider che verrà analizzato successivamente. In seguito, una volta stabilito l'angolo da utilizzare, si calcola l'errore tra l'angolo di riferimento e quello di rotta misurato istante per istante. Questo risultato viene fornito come input alla classe

BallView così che possa essere rappresentato graficamente. Nelle ordinate della visualizzazione con la pallina si trova la differenza tra la quota del target e quella attuale.

4.2 BallView.Java

La classe BallView.java è la classe che si occupa della rappresentazione grafica della pallina che è il fulcro dell'applicazione stessa. Essa, molto semplicemente, viene utilizzata come una funzione di rappresentazione numerica su di un grafico XY perciò, ad ogni chiamata, le viene fornito un valore sull'asse X e uno sull'asse Y che vanno a determinare la posizione della pallina sul piano.

```
public class BallView extends View{
  private double cx;
  private double cy;
  private double radius=20;
  private Paint paint;
```

Il codice si apre con le consuete importazioni scritte in automatico e, successivamente, si inizializza la classe pubblica BallView che contiene il codice della visualizzazione grafica. Tale classe sarà estesa alla classe View che le permette di creare un oggetto grafico che poi viene inserito nella vista principale. In seguito, vengono inizializzate e definite alcune variabili come "cx" e "cy" che identificano la posizione della pallina sul piano, "radius" che determina il raggio della circonferenza e "paint" che serve per il colore.

```
public BallView(Context context){ super(context);}
```

Questo comando viene utilizzato quando la vista viene creata programmaticamente (cioè, utilizzando solo il codice Java e non XML). Il parametro Context context rappresenta l'ambiente in cui la vista viene creata. Il comando super(context) serve per assicurarsi che la vista public class erediti il comportamento di base della classe View.

```
public BallView (Context context, AttributeSet attrs){
  super(context, attrs);
  paint = new Paint();
```

```
    paint.setColor(0xFF0000FF); //colore blu
}
```

Questo costruttore invece, è necessario quando la vista viene creata da un file XML ed ha in ingresso sia il contesto che l'AttributeSet attrs che contiene gli attributi definiti nel file.XML.

Super(context, attrs) chiama il costruttore della classe base View fornendo in ingresso il contesto e gli attributi. Questo permette alla vista di ereditare e utilizzare le proprietà definite nel file XML.

Infine, si definisce un nuovo oggetto paint che è utilizzato per disegnare la vista e, successivamente, si attribuisce al pennello Paint il colore blu, utilizzando il codice esadecimale. Ora, ad ogni chiamata del comando paint, l'elemento sarà di colore blu.

```
public BallView(Context context, AttributeSet attrs, int defStyleAttr){
    super(context, attrs, defStyleAttr);}
```

Questo costruttore viene utilizzato quando la vista viene creata dall'XML e consente di applicare uno stile di default.

defStyleAttr è un attributo di stile che può essere usato per applicare uno stile di default alla vista mentre il comando *super(context, attrs,defStyleAttr)* chiama il costruttore della classe base View, passando il contesto, gli attributi e lo stile di default. Questo permette alla vista di ereditare il comportamento di base e applicare lo stile di default, se specificato.

```
@Override //facciamo un quadrato pefetto
protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec){
    super.onMeasure(widthMeasureSpec, heightMeasureSpec);
    int width= View.MeasureSpec.getSize(widthMeasureSpec); //ottengo le misure suggerite dai genitori
    int height=MeasureSpec.getSize(heightMeasureSpec);
    int size =Math.min(width,height);          //cerco il minimo

    setMeasuredDimension(size,size);
}
```

Questo metodo è stato progettato per fare in modo che l'area in cui si muove la pallina sia un quadrato perfetto della dimensione massima ottenibile in base alle dimensioni della schermata. In particolare, si prelevano in ingresso i due parametri che specificano le dimensioni suggerite dai "genitori" della vista e si richiedono questi

due parametri tramite il comando *super.onMeasure*. Successivamente si definiscono due varabili ("width" ed "height") che conterranno rispettivamente i due parametri richiesti in precedenza così da poterne trovare il minimo e inserirlo nella variabile appena inizializzata, "size". Come ultimo comando si procede a impostare le dimensioni dell'area come un quadrato di lato "size".

```
@Override
protected void onDraw(Canvas canvas){
  super.onDraw(canvas);
  if (cx==0 && cy==0){
    cx=getWidth()/2;
    cy=getHeight()/2;
  }
  canvas.drawCircle((float)cx,(float) cy,(float) radius, paint);
  float width=getWidth();
  float heigth=getHeight();
  canvas.drawLine(width/2, 0, width/2, heigth, paint);
  canvas.drawLine(0, heigth/2, width, heigth/2, paint);
}
```

Questo metodo è il responsabile della vista personalizzata sullo schermo e grazie al comando super, sovrascrive il comando onDraw. Tale comando costruisce il metodo originale per la creazione della vista ogni volta che è necessario. Viene fornito come ingresso il parametro Canvas [14] che rappresenta l'area su cui verrà disegnata la visualizzazione della pallina. In seguito, troviamo una verifica tramite "if" che verifica se "cx" e "cy" sono nulle (cioè non sono state ancora inizializzate) e in questo caso, si ottengono con il comando corrispondente, rispettivamente la larghezza e l'altezza della vista che vengono prima divise per due e poi memorizzate in "cx" e "cy" così che l'origine sia il centro della vista quadrata. Nel caso in cui cx=0 e cy=0, si otterrebbe la posizione in un angolo. Il comando successivo disegna un cerchio sul canvas (area di lavoro) quando gli vengono fornite le coordinate del centro (come variabile float), il raggio del cerchio ed il colore. In questo modo, ad ogni iterazione, si disegna il cerchio nella posizione desiderata semplicemente impostando il corretto dato su "cx" e "cy". L'ultimo blocco di comandi ha lo scopo di disegnare i due segmenti perpendicolari visibili nel rettangolo e la cui intersezione definisce l'origine del sistema. A tal fine sono state inserite in due variabili l'altezza e la larghezza della vista e, in seguito, è

stato eseguito il comando che disegna le linee fornendo il punto iniziale, quello finale ed il colore.

```
public void updatePosition(double error, double altitude){
    // calcola la nuova posizione della pallina
    double width = getWidth();
    double height =getHeight();
    cx=width/2+error*3;
    cy=height/2+(altitude)*3;

    // limitiamo i movimento della pallina ai bordi
    cx=Math.max(radius, Math.min(cx, width-radius));
    cy=Math.max(radius, Math.min(cy, height-radius));

    invalidate(); //richiede ridisegno della vista
  }
} //graffa di chiusura della public class BallView
```

Quest'ultimo blocco ha il compito di aggiornare il valore contenuto nelle variabili "cx" e "cy" ogni qualvolta il comando viene chiamato, così che si possa ridisegnare la pallina nella nuova posizione. Ad ogni chiamata nella classe Main Activity viene fornito in ingresso il dato da rappresentare su ogni asse che, per semplicità, viene chiamato "error" sulle ascisse e "altitude" sulle ordinate. Come eseguito in precedenza, si inseriscono in due variabili l'altezza e la larghezza della schermata. Successivamente si impostano i valori di "cx" e "cy" partendo dal centro della vista (quindi con la larghezza e l'altezza diviso due nelle rispettive variabili) e sommando il valore da rappresentare moltiplicato per un guadagno così da accentuare lo spostamento dal centro.

L'ultima coppia di comandi ha lo scopo di evitare che la pallina esca dai bordi dell'area di rappresentazione, impostando "cx" e "cy" come il loro valore massimo nel caso scaturisse un valore troppo elevato dai calcoli. Il metodo si chiude con la chiamata invalidate(), necessaria per richiedere il ridisegno della vista.

4.3 BeamRider.Java

Nella classe BeamRider.Java viene implementato l'algoritmo di guida descritto al capitolo 4. Si è cercato di utilizzare la stessa denominazione delle variabili e dei vettori così da agevolare la comprensione del codice stesso.

```
public class Beam_rider {
  public static double[] error_calculation(double[] NED, double[] Arr_point, double[] Init_point) {

    double Epsilon_Arr_point = Math.atan2(Arr_point[0]-Init_point[0] , Arr_point[1]-Init_point[1]);
    double Epsilon = Epsilon_Arr_point - Math.atan2(NED[0]-Init_point[0] ,NED[1]-Init_point[1]);


double S = Math.sqrt(Math.pow(NED[0]-Init_point[0], 2) + Math.pow(NED[1]-Init_point[1], 2)) *
Math.sin(Epsilon);

    double[] R_NED = new double[]{
        Arr_point[0] - NED[0],
        Arr_point[1] - NED[1],
    };


    double R = Math.sqrt(Math.pow(R_NED[0], 2) + Math.pow(R_NED[1], 2));
    double ki_target = Math.atan2((Arr_point[1] - NED[1]) , (Arr_point[0] - NED[0]));

   double ki_beam = Math.atan2((R * Math.sin(ki_target) - S * Math.sin(Epsilon_Arr_point)) , (R *
Math.cos(ki_target) + S * Math.cos(Epsilon_Arr_point)));

    double[] C_NED = new double[]{
        R * Math.cos(ki_target) + S * Math.cos(Epsilon_Arr_point),
        R * Math.sin(ki_target) - S * Math.sin(Epsilon_Arr_point),
    };


    double [] angoli_ki=new double[]{
        ki_target *180/Math.PI,
        ki_beam *180/Math.PI,
        R,
    };
    return angoli_ki;
  }
}
```

Come primo passo viene definita la classe pubblica BeamRider e alla riga successiva è definito l'unico metodo (chiamato error_calculation) che conterrà l'intero algoritmo. Il metodo viene definito come pubblico, statico e double e necessita di tre vettori in ingresso forniti alla chiamata del metodo stesso. I vettori in ingresso vengono definiti come double e hanno la stessa denominazione di quelli nella classe MainActivity.Java al fine di semplificarne l'utilizzo ma, quello che è fondamentale, è

l'ordine con cui vengono inseriti nel comando di chiamata e la compatibilità del tipo di variabile.

Nei comandi successivi vengono definite e calcolate le variabili ε (epsilon) in un unico comando inserendo double *nome_variabile=calcoli*. Si segnala che tutti gli strumenti matematici sono all'interno della classe Math perciò, per utilizzare la funzione seno, si utilizzerà il comando Math.sin() e via dicendo. In particolare, le funzioni trigonometriche lavorano in radianti, di conseguenza, è necessario convertire tutte le variabili tramite la funzione Math.toRadian().

Successivamente è definito il vettore R_NED che rappresenta il vettore R in assi NED e, in questo caso, si utilizza un'unica istruzione per definire ed assegnare i valori al vettore ovvero ogni riga tra le parentesi graffe corrisponde una cella dell'array.

Proseguendo nei comandi successivi, vengono definiti ed assegnati i valori di C (sia in modulo che in assi NED), di χd_{target} e χd_{beam} e, infine, si definisce un vettore *angoli_ki* in cui vengono inseriti tutti i dati necessari nel codice esterno alla classe BeamRider.Java.

Il comando *return* è quello che permette di riportare un certo dato al di fuori del metodo ed assegnarlo ad una variabile con il vincolo di poter essere utilizzato per un solo elemento; è per questo che si è reso necessario il vettore *angoli_ki*.

4.4 Converter.Java

La classe Converter.Java è una classe creata appositamente per contenere tutti gli algoritmi di conversione necessari per la riuscita del programma. Al suo interno sono contenuti i metodi per la conversione da LLA a ECEF, da ECEF a NED e per l'angolo di rotta.

```
public class Converter {
  private static final double a=6378137.0;
  private static final  double f= 1/298.257223563;
  private static final   double e_quadro=2*f-f*f;
  private static final String TAG="MainActivity";
```

Definizioni dei parametri necessari durante le conversioni.

```
public static double[] lla_to_ecef (double[] LLA){
   LLA[0]=Math.toRadians(LLA[0]);
   LLA[1]=Math.toRadians(LLA[1]);

   double N=a/(Math.sqrt(1-(e_quadro*Math.sin(LLA[0])*Math.sin(LLA[0]))));
   double [] ECEF=new double[3];
   ECEF[0]=(N+LLA[2])*Math.cos(LLA[0])*Math.cos(LLA[1]);
   ECEF[1]=(N+LLA[2])*Math.cos(LLA[0])*Math.sin(LLA[1]);
   ECEF[2]=(N*(1-e_quadro)+LLA[2])*Math.sin(LLA[0]);
   return ECEF;
}
```

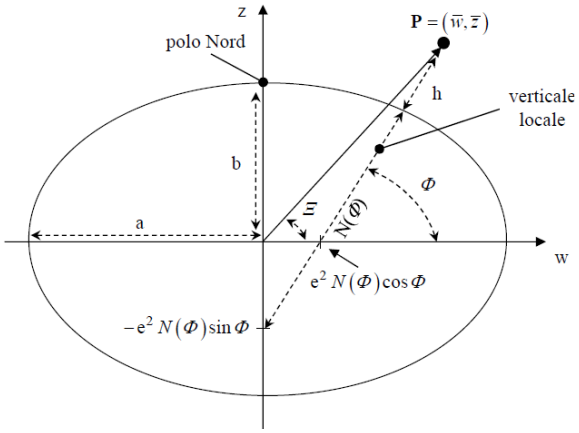

Figura 6. Legame tra il sistema LLA ed il sistema ECEF

Questo metodo converte le coordinate GPS fornite sotto forma di latitudine, longitudine e quota in un sistema in assi ECEF (Earth-Centered Earth-Fixed) [12]. Si ricorda che il vettore in ingresso necessita di coordinate in radianti. Successivamente si individua N che è la retta passante per la verticale locale e che intercetta l'asse di rotazione terrestre. In seguito, si calcola il vettore ECEF nelle sue tre componenti tramite l'opportuna matrice. Come ultimo passo, tramite il comando return(), si riporta nella classe in cui è stata effettuata la chiamata, il vettore ECEF appena calcolato così da poterlo utilizzare in seguito.

```
public static double[] ecef_to_NED (double[] ECEF, double[] ECEF_zero, double[] LLA_zero) {
   double [] DELTA = new double[]{
         ECEF[0]-ECEF_zero[0],
         ECEF[1]-ECEF_zero[1],
         ECEF[2]-ECEF_zero[2],
   };
   Log.d(TAG, "NED Position updated  " + DELTA);
   double [][] ECEF_to_NED_Matrix={
     {-Math.sin(LLA_zero[0]) * Math.cos(LLA_zero[1]), -Math.sin(LLA_zero[1])*Math.sin(LLA_zero[0]),
```

```
Math.cos(LLA_zero[0]) },
     {-Math.sin(LLA_zero[1]), Math.cos(LLA_zero[1]), 0},
     {-Math.cos(LLA_zero[1])*Math.cos(LLA_zero[0]), -Math.cos(LLA_zero[0])*Math.sin(LLA_zero[1]), -
Math.sin(LLA_zero[0])

}
   };
   double [] NED= new double [3];
   for (int i=0; i<3; i++)

{NED[i]=ECEF_to_NED_Matrix[i][0]*DELTA[0]+ECEF_to_NED_Matrix[i][1]*DELTA[1]+ECEF_to_NED_Ma
trix[i][2]*DELTA[2];}
   return NED;
}
```

In questo nuovo metodo si converte il dato dal sistema ECEF a quello NED (North, East, Down), il che rende necessaria la fornitura di un punto specifico in coordinate ECEF e LLA che fungerà da origine del sistema NED.

$$\mathbf{C}_{ECEF}^{NED}(\lambda,\Phi)=\begin{pmatrix}-\sin\Phi\cos\lambda & -\sin\Phi\sin\lambda & \cos\Phi\\ -\sin\lambda & \cos\lambda & 0\\ -\cos\Phi\cos\lambda & -\cos\Phi\sin\lambda & -\sin\Phi\end{pmatrix} \qquad (11)$$

Il vettore denominato DELTA contiene la differenza tra la posizione del dispositivo e quella di origine in assi ECEF. In seguito, si predispone la matrice di conversione da assi ECEF a NED (11) *[15]* così che possa essere moltiplicata per il vettore DELTA al fine di trovare la posizione in assi NED. In Java non è possibile effettuare direttamente la moltiplicazione matrice per vettore, di conseguenza essa viene effettuata tramite un ciclo for. Il metodo si conclude con il return del dato in assi NED così che possa essere utilizzato nel MainActivity.Java.

```
public static double bearing_converter(double ki){
   if (ki>180){
      ki=-(360-ki);
   }
   return ki;
}
```

L'ultimo metodo implementato nella classe Converter.Java è finalizzato a convertire il dato dell'angolo di rotta che è fornito tra 0 e 360° in un angolo definito tra 0 e 180° o tra 0 e -180°. Tale conversione avviene tramite il comando "if" che lascia invariato il

dato se l'angolo di rotta è minore di 180° mentre, se dovesse essere maggiore di 180°, il dato dell'angolo di rotta diventa uguale a –(360-angolo_di_rotta).

4.5 Compass.Java

La classe Compass.Java è una classe basata su BallView.Java che permette di disegnare una pallina che si muove su di una bussola al fine di indicare la direzione del vettore velocità.

```
private double cx;
private double cy;
private double radius=20;
private float radius_compass=150;
private Paint paint;
private Paint black;
private Paint white;
public Compass(Context context){ //serve per implementare la vista quando è creata
programmaticamente cioè usando Java ma senza XML
   super(context);
}

//public Square_View(Context context, AttributeSet attrs){ //costruisce la vista quando parto da un
XML è quindi preleva gli attributi Xml
//super(context, attrs);}
public Compass (Context context, AttributeSet attrs){
   super(context, attrs);
   paint = new Paint();
   paint.setColor(0xFF0000FF); //colore blu
}

public Compass(Context context, AttributeSet attrs, int defStyleAttr){ //Questo costruttore è utilizzato
quando la vista viene creata da XML e consente anche di applicare uno stile di default. defStyleAttr è
un attributo di stile che può essere usato per applicare uno stile di default alla vista.
   super(context, attrs, defStyleAttr); //super passa i parametri al costruttore della classe base (view in
questo caso)
}
@Override //facciamo un quadrato pefetto
protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec){
   super.onMeasure(widthMeasureSpec, heightMeasureSpec); //sostanzialmente andiamo a
chiamare la classe base per ottenere le misure suggerite dai genitori
   int width= View.MeasureSpec.getSize(widthMeasureSpec); //ottengo le misure suggerite dai genitori
   int height=MeasureSpec.getSize(heightMeasureSpec);
   int size =Math.min(width,height); //cerco il minimo
```

```
    setMeasuredDimension(size,size);
}
```

Il codice è uguale a quello della BallView fino al metodo OnDraw se non per la definizione di altri tre colori che verranno poi utilizzati nel codice. La variabile compass_radius rappresenta il raggio esterno della bussola che poi verrà usato come riferimento per il posizionamento di altri oggetti all'interno della bussola stessa.

```
@Override
protected void onDraw(Canvas canvas){
    super.onDraw(canvas);
    float width=getWidth();
    float heigth=getHeight();
    black = new Paint();
    black.setColor(0xFF000000); //colore nero
    white = new Paint();
    white.setColor(0xFFFFFFFF); //colore blu
```

Questo primo blocco di comandi è molto simile a quello della BallView nella classe onDraw se non per l'assegnazione del colore alla variabile corrispondente, tramite il relativo codice esadecimale.

```
    canvas.drawCircle(width/2, heigth/2, radius_compass,black);
    canvas.drawCircle(width/2, heigth/2, radius_compass-2,white);
    canvas.drawCircle((float)cx,(float) cy,(float) radius, paint);
    canvas.drawCircle(width/2,heigth/2,10, paint);
```

Successivamente, vengono disegnati due cerchi concentrici: uno nero, con la dimensione massima consentita e uno bianco con un raggio inferiore di 2dp. Questo crea una circonferenza nera di 2dp di spessore, che funge da perimetro della bussola. Il comando *canvas.drawCircle* consente di disegnare un cerchio specificando come parametri la posizione del centro (nelle due coordinate), il raggio e il colore desiderato.

```
    black.setTextSize(50);

    canvas.drawText("N", width/2-18, heigth/2-radius_compass+50, black);
    canvas.drawText("S", width/2-16, heigth/2+radius_compass-20, black);
    canvas.drawText("E", width/2+radius_compass-50, heigth/2+18, black);
    canvas.drawText("W", width/2-radius_compass+18, heigth/2+18, black);          }
```

In questo blocco vengono scritte nella posizione corretta le quattro lettere che caratterizzano la bussola, solamente dopo aver settato la dimensione del testo a

50dp. Il comando DrawText necessita, in ordine, del testo da scrivere tra virgolette, della posizione in cui scriverlo sia lungo l'orizzontale che lungo la verticale e del colore che, in questo caso, è contenuto nella variabile black.

```
public void updatePosition(double ki){
    // calcola la nuova posizione della pallina
    double width = getWidth();
    double height =getHeight();
    cx=(width/2)+(radius_compass-30)*Math.cos(Math.toRadians(ki-90));
    cy=(height/2)+(radius_compass-30)*Math.sin(Math.toRadians(ki-90));
     invalidate(); //richiede ridisegno della vista   }
} //parentesi per la chiusura della public class Comass.Java
```

L'ultimo blocco di codice di questa classe è un metodo che viene chiamato dalla Classe MainActivity e che aggiorna costantemente le variabili "cx" e "cy" che rappresentano la posizione della pallina. Come già descritto in precedenza, si inseriscono in due variabili l'altezza e la larghezza del layout utilizzabile e successivamente si aggiorna la posizione "cx" e "cy" partendo dal centro e sommando la posizione della pallina. Tale posizione è definita da una distanza dal centro costante e dall'angolo di rotta misurato dal dispositivo (quindi in coordinate polari), convertite in coordinate cartesiane. L'angolo di rotta (χ) è l'unico input di questa funzione. Come in ballView.java, il codice si chiude con il comando invalidate() utile per richiedere il ridisegno della vista.

4.6 ActivityMain.xml

ActivityMain.xml è la parte del codice dell'applicazione che si occupa dell'aspetto grafico della visualizzazione. Le informazioni vengono prelevate dai file.Java e inserite nella schermata con le opportune modifiche.

Figura 7. Interfaccia grafica dell'applicazione

Il codice adotta un'architettura modulare annidata di LinearLayout, in cui i singoli layout sono disposti all'interno di un sistema di livelli gerarchici in modo tale che ogni modulo ne contenga un altro. Nel livello più esterno si trova un LinearLayout che a sua volta contiene due layout differenti: ovvero la BallView e un altro LinearLayout. Quest'ultimo contiene la parte restante delle informazioni che si vogliono visualizzare numericamente. Si ricorda che in ogni blocco di codice è fondamentale l'id che identifica il blocco stesso e permette la comunicazione con il codice java dell'applicazione.

```
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="horizontal">
```

Il LinearLayout è un layout che organizza i suoi "figli" in una singola colonna o riga. In questo caso, orientation="horizontal" significa che gli elementi "figli" saranno disposti orizzontalmente, uno accanto all'altro. Il comando xmlns:android è

l'attributo di spazio dei nomi XML necessario per permettere al sistema di riconoscere gli attributi specifici di Android all'interno del layout stesso, mentre layout_width="match_parent" e layout_height= "match_parent" fanno sì che il LinearLayout occupi tutta la larghezza e l'altezza disponibili dello schermo.

```
<com.example.test_9_luglio.BallView
  android:id="@+id/BallView"
  android:layout_width="0dp"
  android:layout_height="match_parent"
  android:layout_weight="1"
  android:background="#E1D5D5" />
```

In questo blocco si opera sul posizionamento della vista BallView che però viene definita nel codice Java. Si imposta la larghezza della vista pari a 0 e layout_weight =1: ciò significa che questa vista prenderà metà dello spazio orizzontale disponibile (condividendolo con il LinearLayout accanto). Per quanto riguarda invece l'altezza, si fornisce il comando match_parent così che la vista occupi l'intera altezza del LinearLayout. L'ultima riga imposta lo sfondo della vista di colore bianco avorio.

```
<LinearLayout
  android:id="@+id/rightContainer"
  android:layout_width="0dp"
  android:layout_height="match_parent"
  android:layout_weight="1"
 android:background="#F5F5F5"
 android:orientation="vertical" >
```

Proseguendo con il codice, si definisce il contenitore a destra della BallView in cui vengono inseriti tutti i dati necessari. Il layout_width="0dp" e il layout_weight="1" permettono che questo layout occupi l'altra metà dello spazio orizzontale disponibile. Internamente a questo blocco si individuano una serie di informazioni riguardanti il volo che sono state inserite in maniera tale da privilegiare alcuni dati rispetto ad altri.

```
<!—Compass in alto  al centro -->
<com.example.test_9_luglio.Compass
  android:id="@+id/Compass"
  android:layout_width="180dp"
  android:layout_height="180dp"
```

```
    android:layout_gravity="center_horizontal"
    android:layout_marginTop="10dp" />

<!—Speed error Text -->
<TextView
    android:id="@+id/delta_speed_Label"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:layout_marginTop="5dp"
    android:text="SPEED ERROR [m/s]"
    android:textSize="18sp" />

<TextView
    android:id="@+id/delta_speed_TextView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:layout_marginTop="2dp"
    android:textAppearance="?android:attr/textAppearanceLarge" />

<!—Tasto Beam Rider  -->
<Button
    android:id="@+id/Beam_rider_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:layout_marginTop="10dp"
    android:text="Beam rider" />
```

In particolare, nel blocco a destra, si troverà in alto l'indicatore grafico per l'angolo di rotta "compass" mentre, nella posizione sottostante, è presente l'errore di velocità seguito dal bottone per il beam rider. Più precisamente, per quanto riguarda il blocco denominato "Compass", si definisce la dimensione del suo layout come un quadrato di 180dp per lato, centrato orizzontalmente e con un margine dal bordo superiore di 10dp.

In seguito al "Compass" sono presenti le due sezioni relative alla visualizzazione dell'errore di velocità che si ricorda essere la differenza tra la velocità richiesta per il passaggio sul target e la velocità attuale. Il primo blocco riguarda l'etichetta (label) mentre il secondo si riferisce al dato. Le dimensioni di entrambe le sezioni sono definite in base al contenuto dell'informazione rappresentata al loro interno tramite il

comando “wrap_content”. L’ultima parte definisce la posizione del bottone per l’accensione del beam rider.

```
<!—Due colonne  -->
    <LinearLayout
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:orientation="horizontal"
      android:layout_marginTop="20dp"
      android:gravity="center">

      <!—Colonna di sinistra -->
      <LinearLayout
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:orientation="vertical"
        android:gravity="center_horizontal">

        <!-- V_attuale (Ground Speed) -->
        <TextView
          android:id="@+id/V_attuale_Label"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:text="GROUND SPEED [m/s]"
          android:textSize="14sp" />
        <TextView
          android:id="@+id/V_attuale_TextView"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"

android:textAppearance="?android:attr/textAppear
anceLarge"
          android:layout_marginTop="5dp" />

        <!-- Ki (Angolo di rotta) -->
        <TextView
          android:id="@+id/Ki_Label"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:text="HEADING ANGLE [deg]"
          android:textSize="14sp" />
        <TextView
          android:id="@+id/Ki_TextView"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
android:textAppearance="?android:attr/textAppear
anceLarge"
          android:layout_marginTop="5dp" />

        <!-- Altitude -->
        <TextView
          android:id="@+id/altitude_Label"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:layout_marginTop="10dp"
          android:text="ALTITUDE [m]"
          android:textSize="14sp" />
        <TextView
          android:id="@+id/altitude_TextView"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"

android:textAppearance="?android:attr/textAppear
anceLarge"
          android:layout_marginTop="5dp" />

      </LinearLayout>

      <!—Colonna di destra -->
      <LinearLayout
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:orientation="vertical"
        android:gravity="center_horizontal">

        <!—Distanza dal Target -->
        <TextView
          android:id="@+id/Distance_Label"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:layout_marginTop="10dp"
          android:text="DISTANCE TO TARGET [m]"
          android:textSize="14sp" />
        <TextView
          android:id="@+id/Distance_TextView"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
```

```
android:textAppearance="?android:attr/textAppear
anceLarge"
        android:layout_marginTop="5dp" />

    <!—altitude error -->
    <TextView
      android:id="@+id/delta_altitude_Label"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_marginTop="10dp"
      android:text="ALTITUDE ERROR [m]"
      android:textSize="14sp" />
    <TextView
      android:id="@+id/delta_altitude_to_target_TextView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"

android:textAppearance="?android:attr/textAppear
anceLarge"
        android:layout_marginTop="5dp" />

      </LinearLayout>
    </LinearLayout>
  </LinearLayout>
</LinearLayout>
```

Al di sotto di questi elementi vengono definiti due layout (LinearLayout) che dividono lo spazio rimanente in due colonne identiche che contengono le differenti informazioni di volo permettendo una disposizione a doppia colonna.

Ogni dato di volo viene rappresentato con la rispettiva etichetta (Label) che descrive l'informazione visualizzata. In questa visualizzazione si trovano alcune informazioni come la velocità al suolo (Ground Speed), l'angolo di rotta (Heading Angle), l'altitudine (Altitude), la differenza di altitudine tra il target e quella attuale (Delta_Altitude) e la distanza dal target in linea retta (Distance). Tutti i TextView e le relative etichette sono posizionati uno sotto l'altro utilizzando il comando layout_below per creare un layout verticale all'interno del RelativeLayout. Ogni coppia TextView-Label è centrata orizzontalmente per allineare correttamente il testo.

Si sottolinea che questi Linear_layout, che hanno lo scopo di contenere le varie informazioni di volo, vengono creati gli uni dentro gli altri, con una struttura annidata. All'interno della schermata, che è il Linear_layout più esterno, si trovano a sinistra la ball_view e a destra un ulteriore Linear layout. Quest'ultimo, a sua volta, contiene in alto e centrati i tre elementi precedentemente descritti e, al di sotto di essi, trovano posto i Linear_layout delle due colonne che contengono parte delle informazioni di volo.

4.7 Gradle

Una parte importante del codice che non è mai stata nominata ma meritevole di un sottocapitolo è il file Gradle, necessario alla creazione dell'applicazione stessa in quanto automatizza gran parte del processo di costruzione, consentendo di gestire facilmente le dipendenze e di configurare le impostazioni specifiche del progetto.

Questo file definisce le configurazioni principali del progetto Android, gestendo le versioni SDK, le librerie esterne (dipendenze) e le impostazioni di build per creare l'applicazione. Questa parte è generata in automatico alla creazione dell'applicazione e solo in alcuni casi sarà necessario procedere inserendo alcune dipendenze aggiuntive. In questo caso sono state inserite soltanto due librerie ovvero "libs.material" che implementa la libreria Material Design di Google e "com.google.android.gms:play-services-location:21.0.1" che invece implementa i servizi di localizzazione di Google Play Services.

```
plugins { alias(libs.plugins.android.application)}
android {
  namespace = "com.example.test_9_luglio"
  compileSdk = 34
  defaultConfig {
    applicationId = "com.example.test_9_luglio"
    minSdk = 28
    targetSdk = 34
    versionCode = 1
    versionName = "1.0"
    testInstrumentationRunner = "androidx.test.runner.AndroidJUnitRunner"
  }

  buildTypes {
    release {
      isMinifyEnabled = false
      proguardFiles(
        getDefaultProguardFile("proguard-android-optimize.txt"),
        "proguard-rules.pro"
      ) } }
  compileOptions {
    sourceCompatibility = JavaVersion.VERSION_1_8
    targetCompatibility = JavaVersion.VERSION_1_8
  } }

dependencies {
  implementation(libs.appcompat)
  implementation(libs.material)
  implementation(libs.activity)
```

```
    implementation(libs.constraintlayout)
    implementation("com.google.android.gms:play-services-location:21.0.1")
    implementation("androidx.lifecycle:lifecycle-runtime-ktx:2.6.1")
    implementation("androidx.activity:activity-ktx:1.7.2")
    implementation("androidx.fragment:fragment-ktx:1.5.7")
    testImplementation(libs.junit)
    androidTestImplementation(libs.ext.junit)
    androidTestImplementation(libs.espresso.core)
}
```

5. CONCLUSIONI

Concluso il lavoro di “Sviluppo e prototipazione di un sistema di ausilio alla navigazione aerea su piattaforma mobile” si è svolta la fase di test dell’applicazione, seguendo le indicazioni fornite dall’interfaccia grafica progettata, al fine di raggiungere il target nelle due modalità previste e precedentemente descritte.

Non potendo provare tale applicazione nell’ambiente per cui è stata pensata, ovvero su un elicottero, il test è stato svolto a piedi in un parco pubblico inserendo le coordinate GPS di un punto di partenza e di uno di arrivo. Posizionandosi al di fuori della linea di unione dei punti individuati, sono state seguite le indicazioni suggerite dall’interfaccia grafica fino al raggiungimento del target. Inoltre, è stato testato il comportamento del software nel caso in cui si devi dalla traiettoria ottimale suggerita.

Figura 8. Test dell'applicazione. La linea rossa tratteggiata rappresenta la traiettoria ottimale in cui ci si inserisce

A seguito di alcune regolazioni dei parametri del GPS inseriti, il test ha avuto esito positivo.

Si ricordano infine le fasi necessarie per un corretto utilizzo dell'applicativo:

1) installare Android studio per poter modificare il codice dell'applicazione;
2) scegliere il punto di arrivo e di partenza in termini di coordinate geografiche;
3) installare l'applicativo sul dispositivo mobile;
4) attivare l'applicazione e attendere la fine della fase di inizializzazione;
5) posizionarsi in un punto casuale e verificare in entrambe le modalità di utilizzo (target e beam rider) se le indicazioni dell'interfaccia grafica sono corrette ovvero se si raggiunge il waypoint di arrivo.

Si ritiene che l'applicativo, allo stato attuale, non sia assolutamente adatto ad un uso nella realtà, se non in via sperimentale, in quanto risulta troppo complicato impostare le coordinate GPS direttamente dal codice dell'applicazione.

A conclusione del lavoro svolto si elencano di seguito alcune implementazioni che renderebbero il prototipo sviluppato più appetibile e interessante:

- definire un sistema che, all'apertura dell'applicazione, permetta di inserire le coordinate di partenza e di arrivo;
- definire un sistema che dia la possibilità di inserire più waypoint così che l'applicativo cambi automaticamente il punto di destinazione con quello successivo, una volta raggiunto quello precedente;
- Definire un sistema che permetta l'inseguimento di una velocità desiderata (V_d) in termini vettoriali, includendo quindi azimuth, elevazione e modulo. Attualmente, è possibile correggere solamente l'azimuth (χ_d), ma, ottenendo informazioni anche sulla coppia di componenti rimanenti, sarebbe possibile eseguire un inseguimento completo del vettore desiderato tramite la correzione del modulo e dell'angolo di elevazione (γ_d).

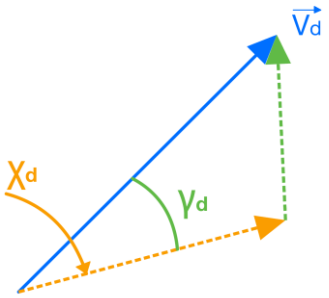

Figura 9. Schematizzazione del Beam Rider per l'inseguimento di un vettore velocità

Dal punto di vista grafico, si potrebbe migliorare il design e i colori dell'interfaccia grafica così da renderla più attraente nonché replicare alcuni strumenti normalmente presenti in cabina di pilotaggio, come l'orizzonte artificiale o l'anemometro.

BIBLIOGRAFIA

[1] J. L. Crassidis, “What Is Navigation?,” *JOURNAL OF GUIDANCE, CONTROL, AND DYNAMICS,* vol. 45, no. 5, 2022.

[2] C. Casarosa, Meccanica del Volo, Pisa University Press, 2010.

[3] X. L. G. Y. Y. Mu, “Advanced Helicopter Cockpit Ergonomic Design Concepts,” in *Complex Systems Design & Management* , Springer, Cham, 2021.

[4] B. Hint, “Whats is an EFB?,” *AircraftIT.*

[5] W. O. Galitz, The essential guide to user interface design: an introduction to GUI design principles and techniques, John Wiley and Sons, 2007.

[6] P. M. Q. A. Guglieri G., Meccanica del volo dell'elicottero: Principi della meccanica e della dinamica del volo, Società Editrice Esculapio, 2019.

[7] M. Stefano, Introduzione alla programmazione con il linguaggio Java, Franco Angeli, 1999.

[8] I. F. Darwin, Android Cookbook: Problems and Solutions for Android Developers, O'Reilly Media, Inc, 2017.

[9] M. N. Marko Magenta, Learning Android: Develop Mobile Apps Using Java and Eclipse, O'Reilly Media Inc..

[10] Z. Paul, Tactical and strategic missile guidance, American Institute of Aeronautics & Astronautics, 1997.

[11] H. E. Jamal, «Progresso e validazione sperimentale di algoritmi GNC applicati ad un rover terrestre,» Tesi di Laurea in Ingegneria Aerospaziale, relatore Emanuele Luigi de Angelis, Università di Bologna, Forlì, 2023.

[12] B. L. a. F. L. L. Stevens, Aircraft Control and Simulation, Hoboken: NJ: John Wiley & Sons.

[13] M. a. A. S. Singhal, «Implementation of location-based services in android using GPS and web services.,» *International Journal of Computer Science Issues (IJCSI),* 2012.

[14] M. N. Tahir, Learning Android Canvas, Packt Publishing, 2013.

[15] P. E. P. Misra, Global Positioning System: Signals, Measurements, and Performance, Ganga-Jamuna Press,, 2009.