

UNIVERSITÀ POLITECNICA DELLE MARCHE
INGEGNERIA INFORMATICA E DELL'AUTOMAZIONE

Stabilità e Imbardata del Quadricottero



Corso di
LABORATORIO DI AUTOMAZIONE
Anno accademico 2024-2025

Studenti:
Alessia Capancioni
Daniele Cota
Nicole Casanova
Naomi Benedetto

Professore:
Andrea Bonci

Dottorandi:
Andrea Serafini
Alessandro Di Biase



Dipartimento di Ingegneria dell'Informazione

INDICE

Introduzione	3
1. Sistema e Modello Matematico.....	4
1.1. Struttura del quadricottero.....	4
1.2. Controllo dell'assetto	6
1.3. Modello Matematico	8
1.3.1. Sistemi di Riferimento.....	8
1.3.2. Caratterizzazione Dinamica	13
1.4. Modello di Controllo	15
1.4.1. Ingressi virtuali e modellazione comandi	15
1.4.2. Calcolo velocità angolari dagli ingressi virtuali	16
1.4.3. Calcolo coefficienti aerodinamici	17
2. Componenti Hardware	19
2.1. Scheda NUCLEO	19
2.2. Power Board	20
2.3. IMU	20
2.4. Motori.....	21
2.5. Batteria.....	23
2.6. Radiocomando.....	24
2.7. ESC.....	25
2.7.1. Programmazione ESC	27
2.7.2. Calibrazione e Avvio ESC	28
2.8. Schema dei collegamenti.....	32
3. Sviluppo Software.....	33
3.1. Periferiche principali	33
3.2. Diagrammi di flusso	34
3.2.1. Diagramma di flusso degli stati del sistema.....	34
3.2.2. Diagramma di flusso del ciclo while.....	35
3.3. Gestione componenti.....	36
3.3.1. Gestione TIM1	36
3.3.2. Gestione IMU	37
3.3.3. PWM	43

3.3.4. Radiocomando	47
3.3.5. PID	53
3.4. Funzionamento complessivo.....	58
3.4.1. Inizializzazione delle periferiche	58
3.4.2. Funzionamento del Codice all'interno del while.....	59
3.4.3. Funzioni di Controllo dei Motori	62
3.4.4. Stima del coefficiente di drag.....	66
4. Test e Simulazioni.....	68
4.1. Codice MATLAB	68
4.2. Processo di taratura	71
4.3. Taratura PID di Roll e Pitch	75
4.4. Taratura PID di Yaw	77
Conclusione e Osservazioni finali.....	80

Introduzione

In questo progetto ci siamo concentrati principalmente sull'ottimizzazione del controllo creato in precedenza per stabilizzare il quadricottero e sull'implementazione del movimento di imbardata.

In particolare, abbiamo aggiunto un altro PID, sia per stabilizzare meglio il drone sul piano xy che per controllare la sua rotazione intorno all'asse z. A tal proposito è stata abilitata la leva principale sinistra del radiocomando, in modo da comandare attraverso il suo stato una rotazione di un certo angolo in senso orario o antiorario.

Questa relazione descrive nel dettaglio il lavoro svolto, a partire dall'analisi della struttura fisica del quadricottero e dalla modellazione matematica dei suoi movimenti. Successivamente vengono illustrati tutti i componenti hardware utilizzati, seguiti dallo sviluppo software, con particolare attenzione alla gestione dei sensori, dei motori e del sistema di controllo PID. La parte finale è dedicata ai test e alle simulazioni che abbiamo eseguito per verificare il comportamento del sistema, approfondendo anche il processo di taratura dei controllori e analizzando le prestazioni complessive raggiunte.

1. Sistema e Modello Matematico

In questa prima parte della relazione si andrà ad analizzare la struttura del quadricottero, il modello matematico che ne descrive il comportamento e il sistema di controllo che ne regola il movimento. Verranno approfonditi i principi fisici alla base del volo, il ruolo delle eliche nella generazione di forza e momento, e le strategie di controllo impiegate per ottenere stabilità e manovrabilità.

1.1. Struttura del quadricottero

I droni quadricotteri rappresentano una delle soluzioni più diffuse nel campo degli aeromobili a pilotaggio remoto, grazie alla loro elevata stabilità, manovrabilità e compattezza. I modelli più recenti, spesso di dimensioni ridotte, sono progettati per operare in ambienti sia interni che esterni, grazie all'integrazione di sensori e sistemi di controllo avanzati.

Il quadricottero, o quadrirotore, è un velivolo dotato di quattro rotori con eliche disposte orizzontalmente, equidistanti e simmetriche rispetto al centro del corpo. La semplicità meccanica, unita alla possibilità di controllare tutti i gradi di libertà attraverso la sola variazione della velocità dei rotori, ne fa una delle configurazioni più diffuse tra i droni.

I rotori con le rispettive eliche sono montati in modo da formare due coppie controrotanti: due ruotano in senso orario (1 e 3) e due in senso antiorario (2 e 4), come mostrato nella figura sottostante (Figura 1.1).

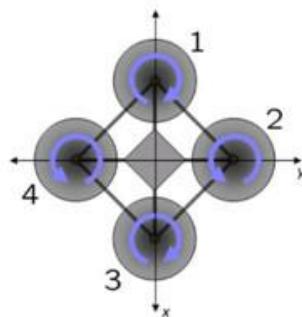


Figura 1.1: Verso di rotazione delle eliche

Il quadricottero ha una struttura simmetrica, con un telaio a forma di croce in cui ogni braccio ospita un motore e un'elica.

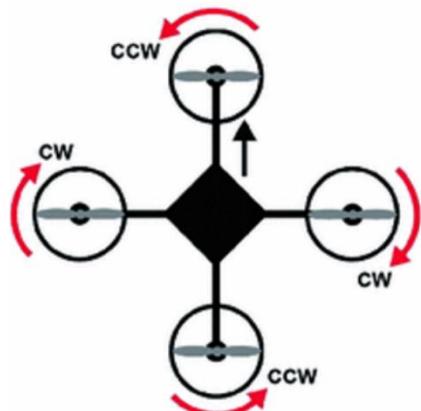
Il corpo centrale contiene l'alimentazione e l'elettronica di controllo, e nel nostro caso include i seguenti componenti principali:

- **Scheda STM32 H745 NUCLEO-144** situata nella parte superiore, che funge da unità di controllo principale.
- **Batteria** da 14.8V
- **Power Distribution Board** collegata ai quattro ESC dei motori, che gestisce la distribuzione efficiente e sicura dell'alimentazione ai vari componenti.
- **Sensore IMU** che rileva l'orientamento e fornisce dati essenziali per la stabilizzazione del drone.
- **Ricevitore** del Radiocomando per il controllo remoto del drone.

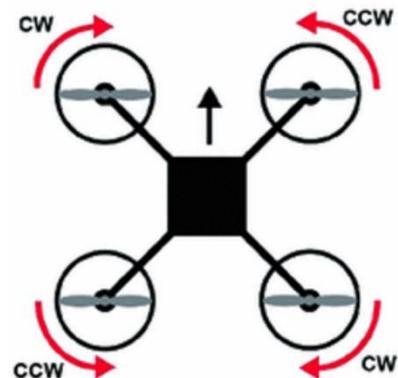
La struttura è progettata per essere leggera ma resistente, in modo da garantire efficienza aerodinamica, manovrabilità e protezione in caso di urti.

Esistono due configurazioni principali per la disposizione dei motori del quadricottero, come mostrato nelle immagini seguenti (Figura 1.2):

- **Configurazione a “+”** : i motori sono allineati lungo gli assi x e y.
- **Configurazione a “X”** : i motori sono disposti lungo le diagonali del piano cartesiano.



(a) Configurazione a +



(b) Configurazione a x

Figura 1.2: Configurazione del quadricottero

La principale differenza tra le due configurazioni riguarda l'orientamento della direzione di avanzamento rispetto alla struttura del telaio e il modo in cui le forze vengono distribuite durante il volo. Queste variazioni influenzano la dinamica del drone e la sua risposta ai comandi, determinando differenze in termini di stabilità e manovrabilità.

1.2. Controllo dell'assetto

Ogni rotore genera una forza di spinta verticale (F), una coppia torcente (M) attorno al proprio asse di rotazione e una resistenza aerodinamica (d) nella direzione opposta al moto del velivolo.

Altitudine: quando tutti i rotori ruotano alla stessa velocità angolare, le forze di spinta si bilanciano e le coppie torcenti si annullano a vicenda, producendo una forza netta verso l'alto senza nessuna rotazione attorno agli assi. L'altitudine del drone (altezza rispetto all'asse z) si controlla quindi aumentando o diminuendo uniformemente la velocità di tutti e quattro i motori.

Il controllo della rotazione del quadricottero attorno ai suoi tre assi viene realizzato variando la velocità angolare dei motori in base al tipo di movimento desiderato:

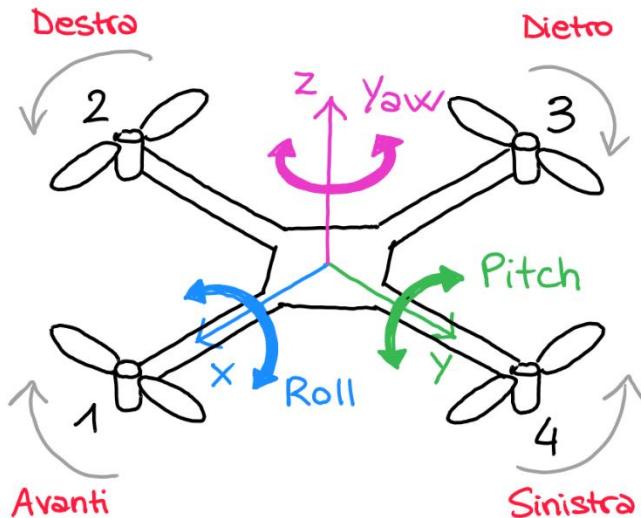


Figura 1.3: Diagramma delle rotazioni del drone rappresentato con la configurazione a “+”

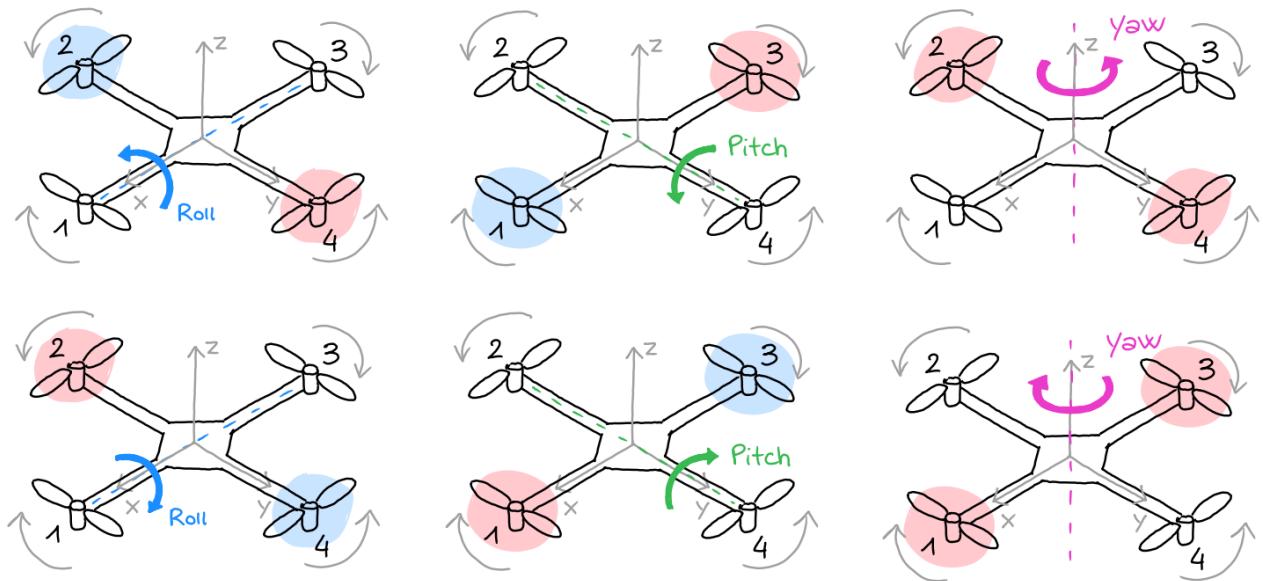


Figura 1.4: Diagramma dei movimenti di *Roll*, *Pitch* e *Yaw* (gli aloni intorno alle eliche rappresentano l'aumento (rosso) o diminuzione (blu) della velocità angolare)

Rollo (Roll): rotazione attorno all'asse x, ottenuta variando la velocità dei motori situati lungo l'asse y (2 e 4), mantenendo costante la somma delle loro spinte. Inclinazione del drone verso destra o verso sinistra.

Beccheggio (Pitch): rotazione attorno all'asse y, ottenuta modificando la velocità dei motori situati lungo l'asse X (1 e 3), mantenendo costante la somma delle loro spinte. Inclinazione del drone in avanti o indietro.

Imbardata (Yaw): rotazione attorno all'asse z, ottenuta variando la velocità delle coppie di motori che ruotano in senso orario (1 e 3) e antiorario (2 e 4), senza alterare la spinta verticale complessiva.

Questo schema di controllo consente al quadricottero di mantenere un assetto stabile e di rispondere in modo preciso ai comandi di volo, utilizzando esclusivamente variazioni nelle velocità angolari dei rotorri.

1.3. Modello Matematico

1.3.1. Sistemi di Riferimento

Introduciamo due sistemi di riferimento fondamentali per descrivere l'assetto e la posizione del drone, che saranno alla base della definizione delle equazioni di moto del velivolo:

1. **Sistema di riferimento inerziale (E_I)**: è un sistema fisso rispetto alla Terra, utilizzato per descrivere la posizione assoluta del drone nello spazio. Generalmente, ha l'origine posizionata in un punto di riferimento arbitrario sulla superficie terrestre e segue una convenzione cartesiana con assi x , y , z .

$$E_I = [x \ y \ z]$$

2. **Sistema di riferimento solidale al drone (E_B)**: è un sistema di coordinate che si muove insieme al velivolo. L'origine è fissata nel centro di massa del drone, con gli assi allineati alla sua struttura, nel nostro caso secondo la configurazione a “+”.

$$E_B = [x_B \ y_B \ z_B]$$

Questi due sistemi di riferimento sono essenziali per descrivere l'orientamento, la velocità e l'accelerazione del drone, consentendo di formulare le equazioni che ne governano il comportamento dinamico.

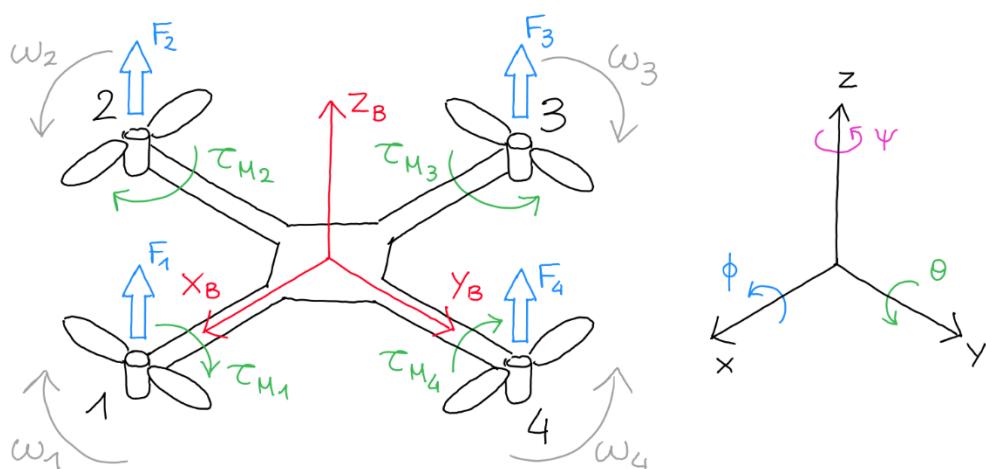


Figura 1.5: Sistemi di riferimento solidale al drone e inerziale.

Dove abbiamo che:

ϕ = angolo di Roll

θ = angolo di Pitch

ψ = angolo di Yaw

F_i = forza di spinta del motore

ω_i = velocità angolare del motore

τ_{Mi} = velocità angolare di drag dei motori, dovuta alla resistenza

aerodinamica (o “drag torque”) che l’aria oppone alla rotazione dell’elica.

Definiamo ora le grandezze fondamentali per descrivere la posizione e l’assetto del drone nel sistema di riferimento inerziale E_I .

- ϵ (**posizione assoluta**): rappresenta la posizione del sistema solidale al drone rispetto al sistema inerziale, determinando le coordinate spaziali del quadricottero.
- η (**assetto del drone**): descrive l’orientamento del sistema solidale al drone rispetto al sistema inerziale attraverso tre angoli di rotazione:
 - ϕ (**roll**): rotazione attorno all’asse x, che determina l’inclinazione laterale (destra - sinistra) del drone.
 - θ (**pitch**): rotazione attorno all’asse y, che indica l’inclinazione avanti - indietro.
 - ψ (**yaw**): rotazione attorno all’asse z.

Questi tre angoli, noti come angoli di Eulero, permettono di descrivere completamente l’orientamento del drone rispetto al sistema di riferimento inerziale e sono essenziali per definire le equazioni che ne regolano il movimento, ottenendo così:

$$\epsilon = [x \ y \ z]^T$$

$$\eta = [\phi \ \theta \ \psi]^T$$

Per descrivere il movimento del drone nelle coordinate del sistema solidale, introduciamo due vettori fondamentali:

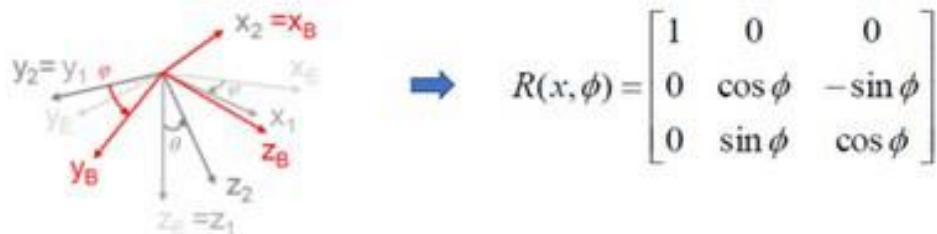
- $\mathbf{V}_B = [u \ v \ w]^T$, che rappresenta le velocità lineari del sistema solidale al drone rispetto al sistema inerziale, espresse nelle coordinate del sistema solidale.
- $\omega_B = [p \ q \ r]^T$, che rappresenta le velocità angolari del sistema solidale al drone rispetto al sistema inerziale, espresse nelle coordinate del sistema solidale.

Si assume che il drone sia perfettamente simmetrico rispetto agli assi x e y . Di conseguenza, gli assi principali d'inerzia coincidono con quelli di simmetria e si ha $I_{xx} = I_{yy}$. La matrice d'inerzia \mathbf{I} assume quindi una forma semplificata:

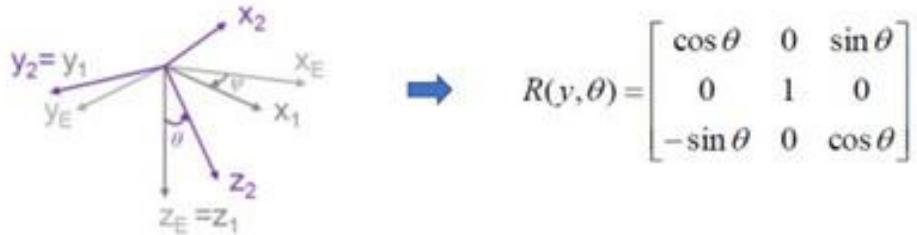
$$\mathbf{I} = \begin{vmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{vmatrix}$$

Per analizzare il comportamento dinamico del drone, utilizzeremo il modello Newton-Eulero, che unisce le leggi del moto di Newton con la teoria dei momenti angolari di Eulero. Questo approccio è utile per descrivere l'evoluzione dell'orientamento e delle forze che agiscono sul drone, tramite l'utilizzo di matrici di rotazione tra i sistemi di riferimento solidale e inerziale:

1. ROLL (rollio): rotazione di un angolo ϕ intorno all'asse x (asse longitudinale, inclinazione destra e sinistra), definito dalla matrice di rotazione $R(x, \phi)$.



2. PITCH (beccheggio): rotazione di un angolo θ intorno all'asse y (asse trasversale, inclinazione in avanti e indietro), definito dalla matrice di rotazione $R(y, \theta)$.



3. YAW (imbardata): rotazione di un angolo ψ intorno all'asse z (asse verticale passante per il baricentro), definita dalla matrice di rotazione $R(z, \psi)$.



L'orientamento del sistema solidale al drone rispetto al sistema di riferimento inerziale è rappresentato tramite una matrice di rotazione ottenuta dalla composizione delle rotazioni attorno agli assi x, y, z. Questa matrice consente di trasformare le coordinate dal sistema solidale al drone (\mathbf{E}_B) a quello inerziale (\mathbf{E}_I) e si costruisce applicando in sequenza le rotazioni legate agli angoli di rollio, beccheggio e imbardata.

$$\mathbf{R} = \mathbf{R}(x, \phi) \mathbf{R}(y, \theta) \mathbf{R}(z, \psi) = \mathbf{R}_\phi \mathbf{R}_\theta \mathbf{R}_\psi = \text{matrice di rotazione da } \mathbf{E}_B \text{ ad } \mathbf{E}_I$$

$$\mathbf{R}(\phi, \theta, \psi) =$$

$$\begin{bmatrix} \cos \psi \cos \theta & \cos \psi \sin \theta \sin \phi - \sin \psi \cos \phi & \cos \psi \sin \theta \cos \phi + \sin \psi \sin \phi \\ \sin \psi \cos \theta & \sin \psi \sin \theta \sin \phi + \cos \psi \cos \phi & \sin \psi \sin \theta \cos \phi - \sin \phi \cos \psi \\ -\sin \theta & \cos \theta \sin \phi & \cos \theta \cos \phi \end{bmatrix}$$

$$\mathbf{E}_B = \mathbf{R}^{-1} \mathbf{E}_I = \mathbf{R}^T \mathbf{E}_I$$

Per quanto riguarda il sistema solidale al drone \mathbf{E}_B , il vettore velocità angolare $\omega_B = [p \ q \ r]^T$, dove p , q ed r indicano rispettivamente le velocità delle rotazioni attorno agli assi x , y e z , può essere scomposto nei contributi delle singole rotazioni di rollio, beccheggio ed imbardata:

$$\omega = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \omega_{roll} + \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \omega_{pitch} + \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \omega_{yaw}$$

Per mettere in relazione le velocità angolari espresse in coordinate del sistema solidale \mathbf{E}_B con le derivate degli angoli di Eulero (ϕ', θ', ψ') espresse in coordinate del sistema inerziale \mathbf{E}_I , si utilizza una matrice di trasformazione T . Questa matrice consente di passare da una rappresentazione all'altra:

$$\begin{bmatrix} \phi' \\ \theta' \\ \psi' \end{bmatrix} = T \begin{bmatrix} p \\ q \\ r \end{bmatrix} \text{ e viceversa } \begin{bmatrix} p \\ q \\ r \end{bmatrix} = T^{-1} \begin{bmatrix} \phi' \\ \theta' \\ \psi' \end{bmatrix}$$

Le espressioni esplicite di T e della sua inversa T^{-1} sono fornite in funzione degli angoli di assetto:

$$T = \begin{bmatrix} 1 & \sin\phi \tan\theta & \cos\phi \tan\theta \\ 0 & \cos\phi & -\sin\phi \\ 0 & \frac{\sin\phi}{\cos\theta} & \frac{\cos\phi}{\cos\theta} \end{bmatrix} ; \quad T^{-1} = \begin{bmatrix} 1 & 0 & -\sin\theta \\ 0 & \cos\phi & \sin\phi \cos\theta \\ 0 & -\sin\phi & \cos\phi \cos\theta \end{bmatrix}$$

Infine, derivando il vettore posizione ϵ e l'assetto η del drone nel sistema di riferimento inerziale \mathbf{E}_I , è possibile descrivere il moto del drone in termini di **velocità lineare assoluta ϵ'** e **velocità angolare assoluta η'** .

Questo avviene mediante l'uso della matrice di rotazione $R(\phi, \theta, \psi)$ che trasforma la velocità lineare \mathbf{V}_B in quella assoluta ϵ' , e della matrice $T(\phi, \theta)$ che trasforma la velocità angolare ω_B in quella assoluta η' :

$$\epsilon' = R \mathbf{V}_B$$

$$\eta' = T \omega_B$$

1.3.2. Caratterizzazione Dinamica

In questa sezione si andrà ad analizzare la dinamica del sistema. Dopo aver descritto le velocità angolari del sistema solidale al drone, esaminiamo l'influenza delle forze esterne \mathbf{F} e dei momenti esterni \mathbf{M} sul suo comportamento.

Dinamica traslazionale

Il moto traslazionale del drone nel sistema di riferimento inerziale è descritto dalla seconda legge di Newton:

$$\sum_{i=1}^N F_i = m \cdot \epsilon'' = S$$

Ogni forza F_i generata da un motore è espressa come:

$$F_i = b \omega_i^2$$

dove ω_i è la velocità angolare del motore e b è il coefficiente di spinta dell'elica.

La somma delle forze generate dai quattro motori produce una spinta complessiva S , diretta lungo l'asse z del sistema solidale al drone $\mathbf{E_B}$:

$$S = \sum_{i=1}^4 F_i = b \sum_{i=1}^4 \omega_i^2 \quad S = \begin{bmatrix} 0 \\ 0 \\ S \end{bmatrix}$$

Dinamica rotazionale

La dinamica rotazionale del drone è descritta dalla legge di Eulero, che lega il momento risultante alle accelerazioni angolari e al momento d'inerzia del motore:

$$M = \sum_{i=1}^N M_i = \eta'' \cdot I$$

La rotazione di ciascuna elica genera un momento torcente M_i attorno al proprio asse, espresso come:

$$M_i = d\omega_i^2 + I_M \eta''_i$$

Dove:

- d è il coefficiente di resistenza aerodinamica dell'elica;
- I_M è il momento d'inerzia del motore;
- η''_i è la velocità angolare dell'elica i .

Nel modello quasi statico, tipico dei controlli di volo a bassa dinamica, si trascura l'effetto di $I_M \eta''_i$, poiché se si hanno motori leggeri I_M è solitamente piccolo, mentre la componente aerodinamica è dominante, per cui si fa la seguente semplificazione:

$$M_i \approx d\omega_i^2$$

Oltre ai singoli momenti M_i , il drone è soggetto a momenti risultanti derivanti dalle coppie di forze generate dai motori, che influenzano l'assetto del drone sui tre assi (Figura 1.4):

- Rollio (ϕ): le forze generate dai motori 2 (asse -y) e 4 (asse +y) agiscono con momenti opposti rispetto all'asse x. Se $\omega_4^2 > \omega_2^2$ il drone compie una rotazione positiva intorno all'asse delle x (rollio verso destra):

$$M_\phi = lb(-\omega_2^2 + \omega_4^2)$$

- Beccheggio (θ): le forze generate dai motori 1 (asse +x) e 3 (asse -x) agiscono con momenti opposti rispetto all'asse y. Se $\omega_3^2 > \omega_1^2$ il drone compie una rotazione positiva intorno all'asse delle y (beccheggio in avanti):

$$M_\theta = lb(-\omega_1^2 + \omega_3^2)$$

- Imbardata (ψ): i motori sono disposti in coppie controrotanti (1-3 orari, 2-4 antiorari) che si sommano algebricamente. Se le coppie torcenti non si bilanciano, si genera una rotazione attorno all'asse z. Nel caso

simmetrico ($\omega_1 = \omega_2 = \omega_3 = \omega_4$), i momenti torcenti si annullano e non si ha nessun movimento di imbardata:

$$M_\psi = d(-\omega_1^2 + \omega_2^2 - \omega_3^2 + \omega_4^2)$$

Il vettore dei momenti torcenti \mathbf{M}_B generati nel sistema solidale E_B è quindi il seguente:

$$\mathbf{M}_B = \begin{bmatrix} M_\phi \\ M_\theta \\ M_\psi \end{bmatrix} = \begin{bmatrix} lb(-\omega_2^2 + \omega_4^2) \\ lb(-\omega_1^2 + \omega_3^2) \\ d(-\omega_1^2 + \omega_2^2 - \omega_3^2 + \omega_4^2) \end{bmatrix}$$

dove l è la distanza tra ciascun motore e il centro di massa del drone, e nel nostro caso vale $l = 0,35$ m.

1.4. Modello di Controllo

1.4.1. Ingressi virtuali e modellazione comandi

Dopo aver analizzato la relazione tra le velocità di rotazione dei motori e i momenti torcenti da essi generati, è ora possibile definire le quattro **uscite virtuali U_i** (Virtual Inputs) che rappresentano i comandi che il sistema di controllo deve generare per posizionare il drone e orientarlo secondo l'assetto richiesto.

Gli ingressi virtuali sono:

- $U_1 \rightarrow$ **Spinta verticale (S)**: forza totale generata dai motori, responsabile della traslazione del sistema solidale E_B lungo l'asse z:

$$U_1 = b(\omega_1^2 + \omega_2^2 + \omega_3^2 + \omega_4^2)$$

- $U_2 \rightarrow$ **Roll (ϕ)**: momento rispetto all'asse x, responsabile del rollio:

$$U_2 = lb(-\omega_2^2 + \omega_4^2)$$

- $U_3 \rightarrow \text{Pitch } (\theta)$: momento rispetto all'asse y, responsabile del beccheggio:

$$U_3 = lb(-\omega_1^2 + \omega_3^2)$$

- $U_4 \rightarrow \text{Yaw } (\psi)$: momento torcente complessivo attorno all'asse z, responsabile dell'imbardata:

$$U_4 = d(-\omega_1^2 + \omega_2^2 - \omega_3^2 + \omega_4^2)$$

Queste relazioni possono essere espresse in forma matriciale come segue:

$$\begin{bmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \end{bmatrix} = \begin{bmatrix} b & b & b & b \\ 0 & -lb & 0 & lb \\ -lb & 0 & lb & 0 \\ -d & d & -d & d \end{bmatrix} \begin{bmatrix} \omega_1^2 \\ \omega_2^2 \\ \omega_3^2 \\ \omega_4^2 \end{bmatrix}$$

1.4.2. Calcolo velocità angolari dagli ingressi virtuali

Poiché i controllori non inviano ai motori direttamente le velocità angolari ω_i , ma i comandi U_i , è necessario invertire la matrice per ottenere i quadrati delle velocità angolari dei motori in funzione dei comandi virtuali:

$$\begin{bmatrix} \omega_1^2 \\ \omega_2^2 \\ \omega_3^2 \\ \omega_4^2 \end{bmatrix} = \begin{bmatrix} \frac{1}{4b} & 0 & -\frac{1}{2lb} & -\frac{1}{4d} \\ \frac{1}{4b} & -\frac{1}{2lb} & 0 & \frac{1}{4d} \\ \frac{1}{4b} & 0 & \frac{1}{2lb} & -\frac{1}{4d} \\ \frac{1}{4b} & \frac{1}{2lb} & 0 & \frac{1}{4d} \end{bmatrix} \begin{bmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \end{bmatrix}$$

1.4.3. Calcolo coefficienti aerodinamici

Per la corretta taratura dei controllori è fondamentale conoscere i coefficienti aerodinamici b e d .

Coefficiente di spinta b

Il coefficiente di spinta dell'elica indica la capacità di ciascuna elica di trasformare la velocità angolare in spinta ed è definito come:

$$b = \frac{S}{\rho A(\omega r)^2}$$

Dove nel nostro caso:

- S = spinta massima generata da un singolo motore [N];
- ρ = 1,225 kg/m³ è la densità dell'aria a livello del mare;
- r = 0,127 m è il raggio dell'elica (lunghezza pala);
- $A = \pi r^2$ è l'area spazzata dall'elica [m²];
- ω = velocità angolare del rotore [rad/s].

S è la spinta totale necessaria a mantenere il drone in volo stabile a mezz'aria. Nel nostro caso però il drone è vincolato a terra tramite il supporto a coppia rotoidale, perciò non genera tutta la spinta necessaria per sollevarsi. Si lavora quindi a basse potenze, ad esempio al 50% di spinta massima.

Basandoci sui valori di S e ω al 50% di spinta massima lasciati dalla relazione precedente,abbiamo che:

$$\begin{aligned} b &= \frac{0,5 \cdot S}{\rho A(\omega r)^2} = \\ &= \frac{0,5 \cdot 31,2N}{1,225kg/m^3 \cdot \pi(0,127m)^2 \cdot (36606rad/s \cdot 0,127m)^2} \\ &\approx 0.00001163 \end{aligned}$$

(Vedere sezione “Conclusione e Osservazioni finali”)

Coefficiente di resistenza aerodinamica d

A differenza di b , il coefficiente di resistenza aerodinamica d è molto più difficile da determinare teoricamente, perché dipende da parametri complessi come la forma delle pale, la viscosità dell'aria e le turbolenze generate. Per questo, abbiamo optato per un approccio sperimentale.

1) Identificazione duty minimo per generare rotazione sullo yaw

Abbiamo realizzato un test in cui è stato incrementato il duty cycle dei motori 1 e 3 fino a generare un cambiamento dello yaw superiore a una soglia (1 grado), mantenendo gli altri due motori costanti.

(Vedere codice e spiegazione nella sezione 3.3.4)

Questo ha permesso di determinare empiricamente il duty minimo per innescare la rotazione, pari a circa 6.1%.

2) Funzione di mapping inversa tra duty e velocità

Dalla funzione di mapping inversa tra duty e velocità è stata recuperata la velocità angolare corrispondente a quel duty (Vedere la funzione di mapping nella sezione “3.2.3 PWM”):

$$\text{duty} = 6.1\% \rightarrow \omega = 423 \text{ rad/s}$$

Da qui, si ottiene:

$$v = \omega r = 423 \text{ rad/s} \cdot 0,127 \text{ m} = 53,72 \text{ m/s}$$

3) Stima teorica del coefficiente d

La formula per il calcolo della resistenza aerodinamica è la seguente:

$$\begin{aligned} d &= \frac{1}{2} \cdot C_d \cdot \rho \cdot A \cdot v^2 = \\ &= \frac{1}{2} \cdot 1,19 \cdot 10^{-10} \cdot 1,225 \text{ kg/m}^3 \cdot \pi(0,127 \text{ m})^2 \cdot (53,72 \text{ m/s})^2 = \\ &\approx 0.0000000107 \end{aligned}$$

Questo valore non è tuttavia risultato coerente in fase di test pratici poiché il drone non riusciva a stabilizzarsi correttamente. Pertanto, abbiamo effettuato una regolazione sperimentale di d (Vedere sezione “Conclusione e Osservazioni finali”):

$$d = 0.000001$$

2. Componenti Hardware

2.1. Scheda NUCLEO

La scheda NUCLEO-144 STM32 H745 ZI Q utilizzata nel nostro progetto è un microcontrollore (MCU) a 144 pin e dual core, cioè con 2 processori ARM Cortex:

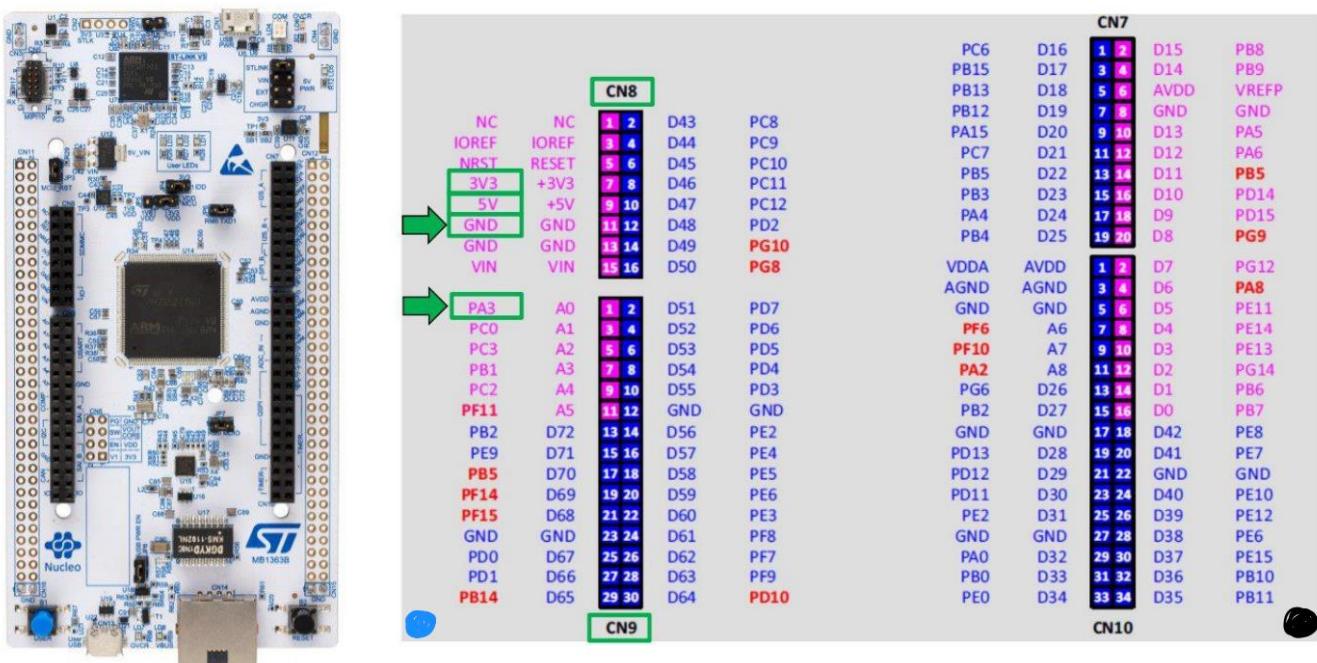
- ARM Cortex-**M7**: a 32 bit, frequenza di clock fino a 480 MHz
- ARM Cortex-**M4**: a 32 bit, frequenza di clock fino a 240 MHz.

L'alimentazione può avvenire:

- tramite USB sul connettore CN1;
- tramite alimentazione esterna da 7-11V tramite pin specifici sul connettore CN8,

con la possibilità di fornire anche 3.3V o 5V come output per alimentare altre schede di espansione.

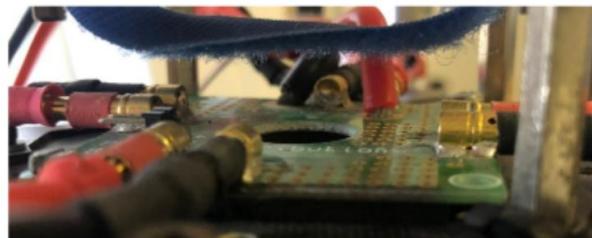
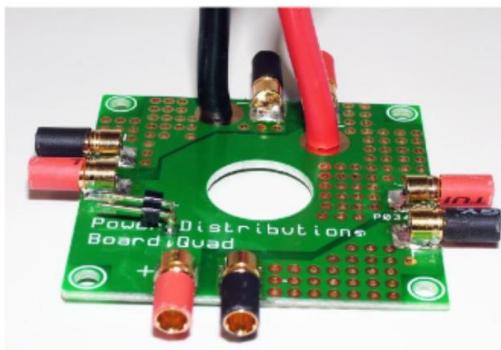
L'MCU è inoltre dotato di un'interfaccia di debugging chiamata ST-LINK V3E, che aiuta a caricare il firmware sulla scheda e a monitorarne l'esecuzione per eventuali errori o problemi.



2.2. Power Board

La Power Distribution Board è un componente fondamentale nei droni, che si occupa di distribuire in modo efficiente e bilanciato la tensione erogata dalla batteria agli altri componenti elettronici.

È progettata con connettori o piste conduttrive che collegano direttamente la batteria agli ESC, garantendo che ciascun ESC riceva la stessa tensione. Questo assicura il funzionamento sincronizzato dei motori, fondamentale per mantenere la stabilità e il controllo del volo.



2.3. IMU

Nel nostro drone è stato impiegato un sensore IMU con chip **BNO055** montato su una scheda ADAFRUIT.

L'IMU (Inertial Measurement Unit) è un dispositivo elettronico utilizzato per misurare e registrare l'**accelerazione lineare** (x, y, z) e l'**accelerazione angolare** (ϕ , θ , ψ) di un oggetto in movimento, integrando sensori come giroscopi e accelerometri.

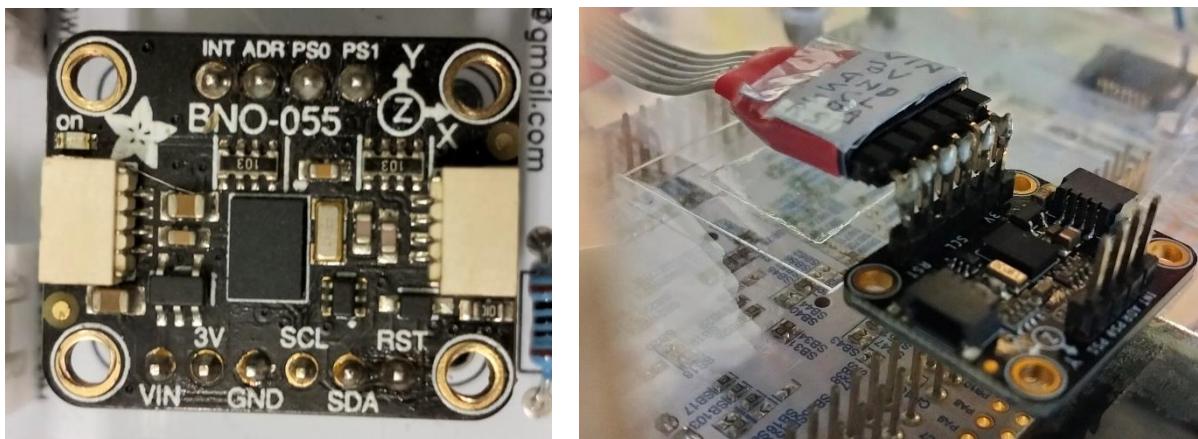
L'IMU utilizzata in questo caso ha le seguenti caratteristiche:

- 9 gradi di libertà;
- **accelerometro** triassiale a 14 bit;
- **giroscopio** triassiale a 16 bit con una gamma di ± 2000 gradi al secondo;
- **sensore geomagnetico** triassiale;
- microcontrollore Cortex M0+ a 32 bit con installato il software Bosch SensorFusion

Sulla scheda ADAFRUIT sono inoltre presenti:

- regolatore di tensione a 3,3 V;
- adattatori di livello per i pin Reset e I2C;
- cristallo esterno da 32.768KHz da utilizzare per ottenere le migliori prestazioni.

Nel nostro progetto quindi l'IMU rileva le accelerazioni lineari sui 3 assi, nonché le velocità angolari di **rollio**, **beccheggio** e **imbardata**, fornendo dati essenziali per il funzionamento dei controllori PID e di conseguenza per la stabilizzazione del drone al variare del suo assetto.



2.4. Motori

Nel drone sono presenti 4 **motori brushless** Turnigy D3536/9 910KV, con delle **Eliche Slow Fly Electric Prop 9057SF** montate in modo da formare due coppie controrotanti. Le principali caratteristiche di questi motori sono:

- classe 35 (diametro 35 mm);
- velocità 910KV: numero di giri del motore al minuto, per ogni volt applicato senza carico.



I motori brushless sono attuatori che convertono energia elettrica in energia meccanica senza l'impiego di spazzole. Sono costituiti da un rotore, composto da magneti permanenti, e da uno statore, sede degli avvolgimenti di rame.

Il funzionamento si basa sulla commutazione elettronica della corrente nelle bobine statoriche, sincronizzata con la posizione del rotore: il campo magnetico generato dalle bobine eccitate attrae o respinge i magneti del rotore, inducendo così la rotazione. La commutazione è gestita dagli ESC, che regolano la sequenza di alimentazione delle bobine per controllare la direzione e la velocità di rotazione.

I motori brushless sono quindi particolarmente adatti per applicazioni che richiedono alta efficienza, affidabilità e bassa manutenzione, come droni e veicoli elettrici.

Tra le principali caratteristiche tecniche dei motori si evidenziano:

- **corrente massima:** massimo flusso di corrente continuo che il motore può sopportare senza surriscaldarsi;
- **tensione massima:** tensione massima ammissibile oltre la quale il motore rischia danni permanenti;
- **resistenza interna:** un valore più basso indica minori perdite di energia e maggiore efficienza complessiva.

Le specifiche dettagliate dei motori utilizzati sono riportate nella seguente scheda tecnica:

KV	910kv
Fasi	3
Massima corrente	25.5A
Potenza massima	370W a 15V
Peso	102g
Batteria	2 ~ 4 Cell /7.4~ 14.8V
Diametro del pozzo	5mm
No corrente do carico	1.5A
Dimensione	35x36mm
Spinta max	1050g
Dimensione prop.	7.4V/12x5 14.8V/10x7
Resistenza interna	0.063 Ohm

2.5. Batteria

Nel drone è montata una batteria FullPower 4s 14.8V 3300mAh, una batteria a 4 celle LiPo:



2.6. Radiocomando

Il radiocomando è utilizzato per armare, avviare e arrestare i motori in sicurezza, evitando il rischio di avvicinare le mani ai rotori in movimento. Il radiocomando è composto da due unità principali:

- **Trasmettitore (Tx):** dispositivo manuale dotato di leve e pulsanti che permettono di controllare parametri come accelerazione, direzione e quota del drone. I comandi impartiti vengono convertiti in segnali elettrici e trasmessi via radio al Ricevitore.
- **Ricevitore (Rx):** installato a bordo del drone, riceve i segnali radio provenienti dal trasmettitore e li converte in segnali elettrici per il controllo diretto del sistema.

Più avanti (*Vedere in “Calcolo fasce di duty con Oscilloscopio” nella sezione 3.3.4*) verrà illustrato il calcolo delle **fasce di duty** cycle del segnale PWM generato dal radiocomando, mediante l'utilizzo dell'oscilloscopio, per analizzare la corretta corrispondenza tra il comando trasmesso e il segnale ricevuto.

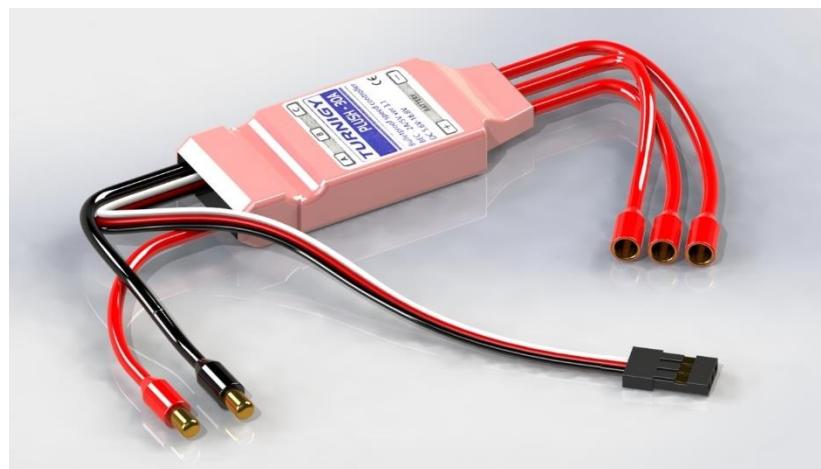


Figura a sinistra: Trasmettitore

Figura a destra: Ricevitore

2.7. ESC

Nel drone sono utilizzati 4 **ESC Turnigy Plush-30A**. Gli ESC (Electronic Speed Controllers) sono dispositivi elettronici che forniscono un controllo preciso della velocità dei motori, gestendo la corrente e la commutazione delle fasi (quindi anche la direzione dei motori) in base ai segnali ricevuti da un microcontrollore o da un radiocomando. Sono posizionati uno per ogni braccio del drone, collegati tutti al scheda STM32.



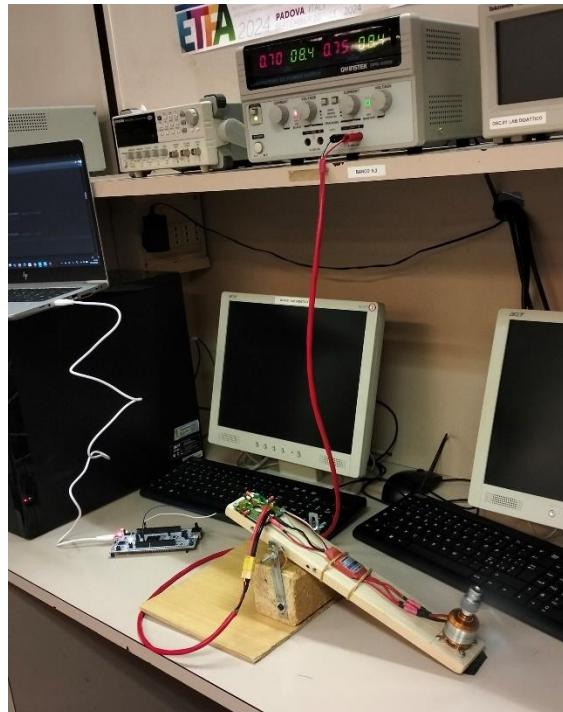
Amperaggio	30 A
Tensione	8.4 – 16.8 V
BEC	5.5 V /4.0 A
Tensione di Cutoff	3.2 V
Peso	24.5g
Dimensioni	36x23.5x10mm
Frequenza	48 MHz

L' ESC utilizzato Turnigy Plush-30A, è progettato per gestire una corrente continua massima di 30 ampere, parametro fondamentale per garantire che il dispositivo possa alimentare il motore elettrico in modo sicuro, senza surriscaldamenti o rischi di danneggiamento. Per quanto riguarda la tensione operativa, questo ESC è compatibile con batterie LiPo da 2 a 4 celle (ossia da 7,4 V a 14,8 V), tipiche dei sistemi di alimentazione per droni. Inoltre, è dotato di un BEC (Battery Eliminator Circuit) integrato, ovvero un circuito che

fornisce una tensione ausiliaria per alimentare componenti come ricevitori e servomeccanismi, eliminando la necessità di una batteria separata per questi elementi.

Per comprendere a fondo il funzionamento dell'ESC utilizzato nel drone, abbiamo inizialmente eseguito una serie di test utilizzando il circuito con singolo braccio disponibile in laboratorio.

Questo era costituito da un'asse di legno su cui erano montati: un ESC, un motore brushless con relativa elica e una powerboard per l'alimentazione dell'ESC tramite un alimentatore esterno.



Per tutte le informazioni riguardanti la programmazione e la calibrazione dell'ESC a cui abbiamo fatto riferimento si può consultare il manuale di utilizzo dell'ESC Turnigy Plush al seguente [link](#).

2.7.1. Programmazione ESC

La programmazione consente di configurare il comportamento dell'ESC prima dell'avvio del motore. Attraverso la Programming Card, è possibile impostare parametri come la modalità di avvio, la frenata, il tipo di batteria e la risposta generale del motore. Questa procedura si esegue senza necessità del radiocomando né del suo ricevitore, poiché lo scopo è solo memorizzare le impostazioni all'interno dell'ESC. Il motore, in questa fase, non verrà avviato.

Per eseguire la programmazione sono necessari solo:

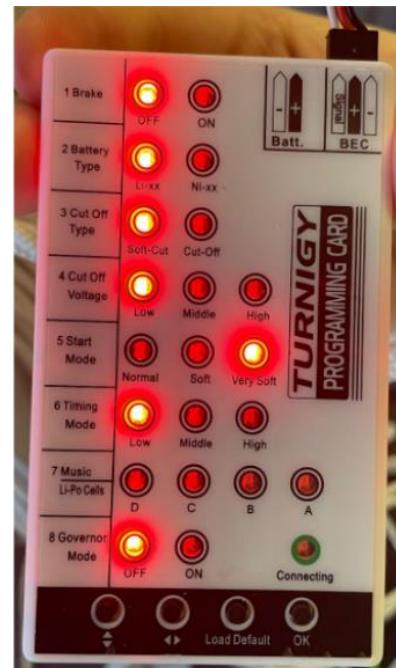
- ESC Turnigy Plush-30A
- Turnigy ESC Programming Card
- Batteria o alimentatore (4,2 V)

Collegare il connettore a 3 pin dell'ESC (quello normalmente collegato alla scheda STM32) direttamente alla programming card, rispettando la corretta orientazione (segnalata sulla card stessa). Dopodiché, collegare l'alimentatore o la batteria all'ESC: l'alimentazione passerà anche alla programming card, che si accenderà automaticamente mostrando le impostazioni attuali memorizzate nell'ESC.

Tra i vari parametri, si consiglia di impostare:

Start Mode → Very Soft (avvio graduale, utile per evitare strappi e picchi di corrente)

Una volta terminata la configurazione, i nuovi parametri verranno salvati automaticamente nell'ESC. A questo punto, è possibile scollegare la programming card e procedere con la calibrazione del segnale PWM.



2.7.2. Calibrazione e Avvio ESC

La calibrazione dell'ESC è una procedura necessaria per far sì che l'ESC riconosca correttamente il range di segnale PWM (duty cycle) entro cui operare. In particolare, questa operazione consente di impostare i valori di duty cycle minimo e massimo che l'ESC userà per controllare il motore, evitando comportamenti anomali o inefficienze.

Questa procedura è fondamentale solo al primo utilizzo dell'ESC o dopo un reset dei parametri, e non è necessaria ad ogni accensione.

Per studiare e testare la calibrazione dell'ESC, è stato utilizzato il circuito a singolo braccio con il seguente setup:

- **Alimentatore** impostato a **8.4 V**, collegato alla Power Board, per simulare una batteria LiPo 2S completamente carica
- **ESC** Turnigy Plush-30A alimentato dalla Power Board
- **Motore brushless** Turnigy D3536/9 910KV, collegato all'ESC
- **ESC** collegato alla scheda STM32 tramite:
 - GND
 - PA6, configurato come uscita PWM tramite TIM3_CH1

La procedura di calibrazione dell' ESC è effettuata in questo modo:

1. Impostare il duty cycle massimo desiderato nel segnale PWM (nel nostro caso 8%, corrispondente a CCR1 = 1600).
2. Collegare l'alimentazione all'ESC solo mentre il segnale PWM è già attivo al valore massimo desiderato. Questo è fondamentale: l'ESC entra in modalità calibrazione solo se all'avvio riceve un segnale PWM con duty cycle superiore al normale range operativo (8-10%). In questa modalità, il primo valore letto viene memorizzato come massimo, mentre il secondo valore successivo viene considerato il minimo accettabile.
3. Verificare che l'ESC emetta il suono di accensione ( 123) seguito da un doppio beep che conferma l'ingresso nella modalità di calibrazione.
4. Lasciare almeno 2s di attesa (nel nostro caso 3s per sicurezza)
5. Impostare il duty cycle minimo desiderato (nel nostro caso 4%, corrispondente a CCR1 = 800).

- Verificare che l'ESC emetta un altro doppio beep, seguito da un beep prolungato: questo segnala che la calibrazione è completata e che l'ESC è in attesa di un segnale valido per iniziare il funzionamento.

Throttle range setting: (Throttle range should be reset when a new transmitter is being used)			
Switch on transmitter, move throttle stick to top	Connect battery pack to ESC, and wait for about 2 seconds	"Beep-beep-" tone should be emitted, means throttle range highest point has been correctly confirmed	Move throttle stick to the bottom, several "beep-" tones should be emitted, presenting the quantity of battery cells

Terminata la procedura di calibrazione, abbiamo consultato ulteriormente il manuale dell'ESC per il corretto avvio:

- All'accensione, l'ESC si aspetta di ricevere immediatamente un segnale PWM con duty cycle pari al valore minimo impostato in calibrazione. In caso contrario, il dispositivo entrerà in errore o tenterà nuovamente (erroneamente) la calibrazione.
- Solo dopo aver riconosciuto il duty cycle minimo, è possibile aumentare gradualmente il segnale PWM fino al valore desiderato, rimanendo sempre all'interno del range precedentemente definito.

Normal startup procedure:

Switch on transmitter, move throttle stick to bottom	Connect battery pack to ESC, special tone like "♪123" means power supply is OK	Several "beep-" tones should be emitted, presenting the quantity of lithium battery cells	When self-test is finished, a long "beep----" tone should be emitted	Ready to go flying now
--	--	---	--	------------------------

Codice

- **Configurazione nel file .ioc**

Attivare **TIM3** (per M4) in modo che il codice funzioni ad una frequenza di 50Hz (20 ms).

Non modificare nulla su Clock Configuration ma imposta il Prescaler in modo che $f_{ck}/(PR + 1) = 1 \text{ MHz}$:

$$UEV = \frac{f_{ck}}{(PR + 1)(PE + 1)} = \frac{64 \cdot 10^6 \text{ Hz}}{64 \cdot 20000} = 50 \text{ Hz}$$

- Clock Source -> Internal Clock
- Channel1 -> PWM Generation CH1 (**PA6** in **TIM3_CH1**)

Su Parameter Settings:

- Prescaler = 64-1
- Counter Period = 20000-1

- **Codice nel main.c (Calibrazione ESC)**

Codice per calibrare l'ESC mandando in sequenza 2 segnali PWM con il valore prima massimo e poi minimo del duty:

```
/* USER CODE BEGIN 2 */
HAL_TIM_PWM_Start(&htim3, TIM_CHANNEL_1);
/* USER CODE END 2 */
/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
    /* USER CODE END WHILE */
    /* USER CODE BEGIN 3 */
    HAL_Delay(2000);
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0, GPIO_PIN_SET); //Accende il Led verde
    //Prima si imposta il PWM al duty cycle massimo che si vuole assegnare
    TIM3->CCR1 = 1600; // Max duty: 8%
    //Si collega l'ESC all'alimentazione mentre il PWM è al valore Max
    //l'ESC emetterà il suono di accensione (B123) seguito da un doppio beep
```

```

//Lasciare almeno 2s di attesa:
HAL_Delay(3000);
HAL_GPIO_WritePin(GPIOE, GPIO_PIN_1, GPIO_PIN_SET); //Accende il Led giallo
//Poi si imposta il PWM al duty cycle minimo che si vuole assegnare
TIM3->CCR1 = 800; // Min duty: 4%
HAL_Delay(3000);
//l'ESC emetterà 3 beep seguiti da un beep più lungo
//l'ESC resta poi in attesa di un segnale (beep ogni 2 secondi)
while (1) //Loop infinito di attesa
{
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_14, GPIO_PIN_SET); //Accende il Led rosso
}
}
/* USER CODE END 3 */

```

- **Codice nel main.c (Avvio ESC)**

Codice per avviare l'ESC prima mandandogli un PWM col minimo duty e poi un altro con un duty a piacere, rimanendo sempre all'interno del range consentito dalla calibrazione:

```

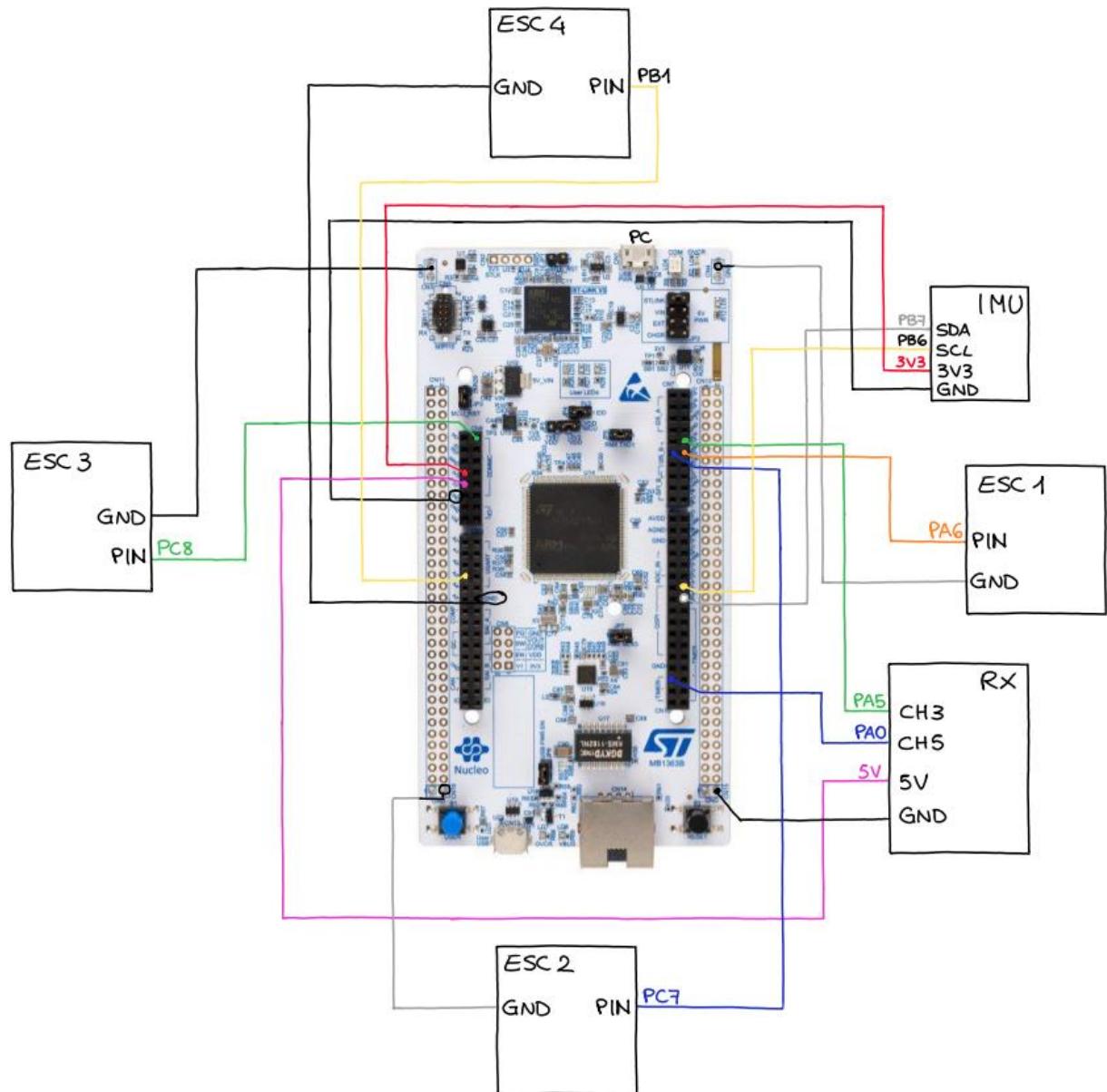
while (1)
{
    /* USER CODE END WHILE */
    /* USER CODE BEGIN 3 */
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0, GPIO_PIN_SET); //Led verde ON
    //Imposta il PWM al duty cycle minimo per avviare correttamente l'ESC
    TIM3->CCR1 = 800; // Min duty: 4%
    HAL_Delay(2000);
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0, GPIO_PIN_RESET); //Led verde OFF
    //Ora si può aumentare gradualmente il duty per controllare il motore

    while (1)
    {
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_14, GPIO_PIN_SET); //Led rosso
        TIM3->CCR1 = 1100; //duty: 5.5%
    }
}
/* USER CODE END 3 */

```

2.8. Schema dei collegamenti

Schema principale dei collegamenti dei 4 ESC, del sensore IMU e del Ricevitore del radiocomando alla scheda STM32:



3. Sviluppo Software

In questa sezione viene descritto in dettaglio il codice sviluppato per il controllo del drone, realizzato all'interno dell'ambiente di sviluppo STM32CubeIDE (versione 1.13.2). Il progetto è stato strutturato in più moduli per gestire separatamente le principali funzionalità: acquisizione dati dalla IMU, controllo PID, generazione e ricezione di segnali PWM per i motori e il radiocomando, e gestione dello stato del sistema.

3.1. Periferiche principali

Le principali periferiche hardware configurate e utilizzate nel codice sono:

- **USART3**: impiegata per la comunicazione seriale verso il PC, utile per il debug e la stampa dei dati acquisiti;
- **I2C1**: utilizzata per la comunicazione con il sensore IMU, configurato in modalità Polling per leggere gli angoli di Eulero (roll, pitch, yaw).
- **TIM1**: configurato come timer base per generare un interrupt periodico che scandisce il tempo di campionamento e aggiorna il ciclo di controllo nel while(1).
- **TIM3**: timer utilizzato per generare segnali PWM verso gli ESC dei motori tramite i canali PWM hardware, controllando la velocità di rotazione.
- **TIM2 e TIM5**: configurati in modalità input capture per ricevere i segnali PWM dal radiocomando:
 - TIM2 gestisce il canale della leva di yaw, utilizzata per comandare la rotazione del drone sull'asse z.
 - TIM5 controlla i segnali di armamento, esecuzione e spegnimento dei motori.

3.2. Diagrammi di flusso

3.2.1. Diagramma di flusso degli stati del sistema

Il diagramma di flusso riportato di seguito rappresenta la logica di gestione degli stati del motore, controllati attraverso la variabile i . Questa variabile assume valori compresi tra 0 e 2, ed è aggiornata in tempo reale in base ai segnali PWM ricevuti dal radiocomando tramite la periferica TIM5:

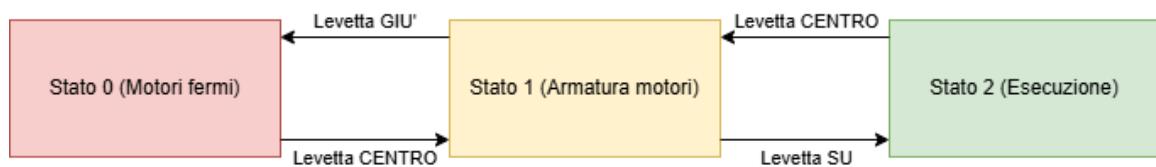


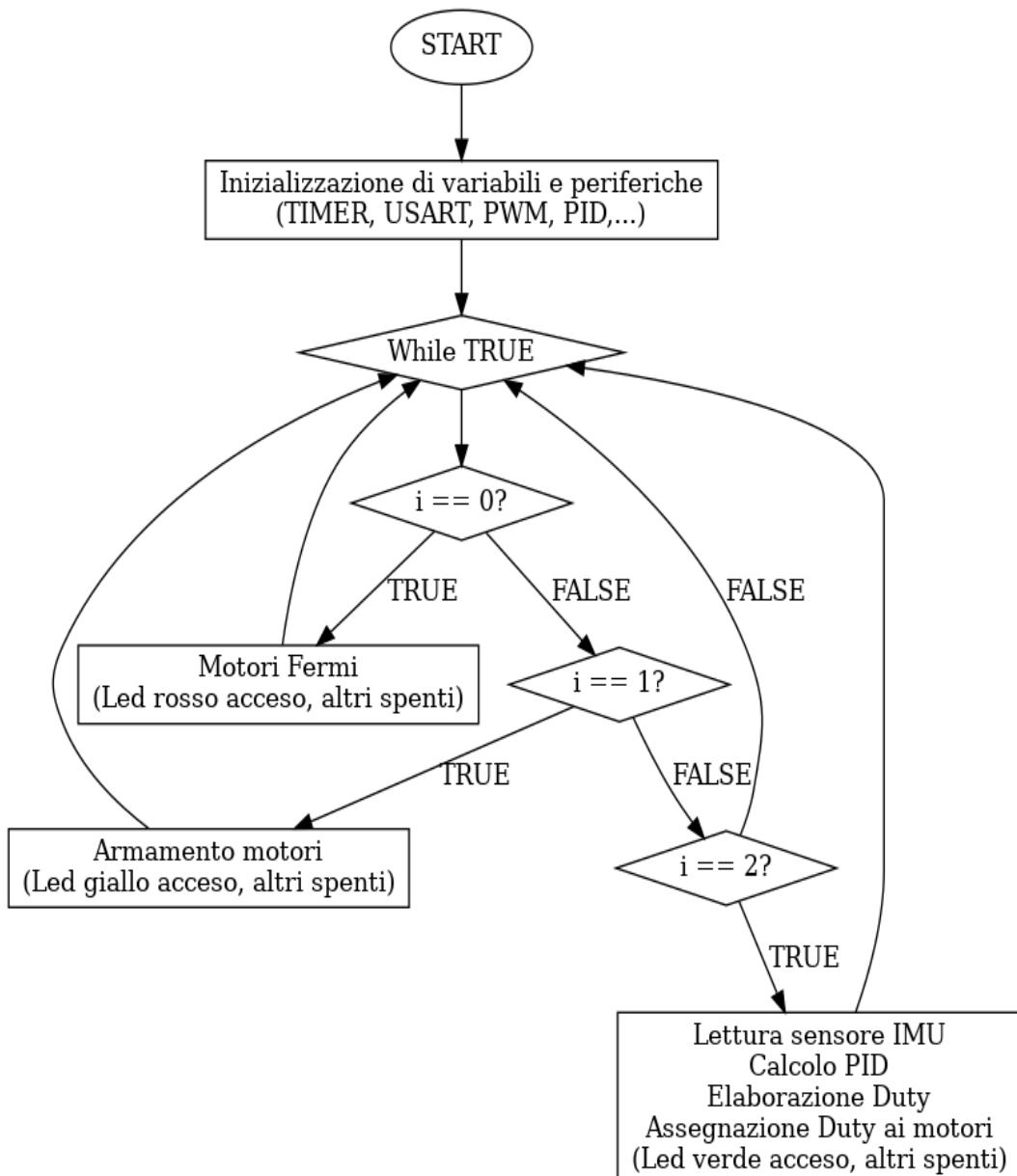
Figure 3.1: Diagramma di flusso degli stati della i

Il passaggio da uno stato all'altro è determinato dalla posizione di un interruttore del radiocomando, situato in alto a sinistra e corrispondente al channel 5 del ricevitore montato sul drone. Il sistema interpreta i comandi in base al duty cycle del segnale in ingresso, aggiornando di conseguenza la variabile di stato.

Questo meccanismo a stati rende il comportamento del drone più sicuro e modulare, separando chiaramente le fasi di avvio, preparazione e funzionamento attivo.

3.2.2. Diagramma di flusso del ciclo while

Il diagramma di seguito riportato mostra, attraverso una semplificazione, come avviene il cambiamento degli stati del sistema e l'esecuzione delle relative operazioni.



3.3. Gestione componenti

Per quanto riguarda la parte software, questa si divide nell'implementazione di, in ordine cronologico, comunicazione con l'IMU, gestione degli ESC con il PWM e taratura dei PID. Otteniamo così un sistema in grado di leggere gli angoli ed elaborare la risposta con lo scopo di regolare la velocità dei motori.

3.3.1. Gestione TIM1

Come primo passo, abbiamo configurato il timer TIM1 in modalità interrupt, per generare un'interruzione ogni 10 millisecondi, corrispondente a una frequenza di 100 Hz. In questo modo il codice viene eseguito a intervalli regolari.

The image consists of two screenshots of the STM32CubeMX software interface, specifically for configuring the TIM1 timer.

Top Screenshot (Mode Tab):

- Runtime contexts:** Cortex-M7 (unchecked), Cortex-M4 (checked), PowerDomain D2 (checked).
- Slave Mode:** Disable.
- Trigger Source:** Disable.
- Clock Source:** Internal Clock.
- Channel Settings (Channels 1-6):** All set to Disable.
- Combined Channels:** Disable.

Bottom Screenshot (Configuration Tab):

- Reset Configuration:** Parameter Settings, User Constants, NVIC Settings, DMA Settings.
- Configure the below parameters:**
- Counter Settings:**
 - Prescaler (PSC - 16 bits value): 64-1
 - Counter Mode: Up
 - Counter Period (AutoReload Register...): 10000-1
 - Internal Clock Division (CKD): No Division
 - Repetition Counter (RCR - 16 bits val...): 0
 - auto-reload preload: Disable
- Trigger Output (TRGO) Parameters:**
 - Master/Slave Mode (MSM bit): Disable (Trigger input effect not delayed)
 - Trigger Event Selection TRGO: Reset (UG bit from TIMx_EGR)
 - Trigger Event Selection TRGO2: Reset (UG bit from TIMx_EGR)

TIM1 Mode and Configuration				
Configuration				
<input type="button" value="Reset Configuration"/> <input checked="" type="checkbox"/> Parameter Settings <input checked="" type="checkbox"/> User Constants <input checked="" type="checkbox"/> NVIC Settings <input checked="" type="checkbox"/> DMA Settings				
NVIC2 Interrupt Table	Enabled	Preemption Priority	Sub Priority	
TIM1 break interrupt	<input type="checkbox"/>	0	0	
TIM1 update interrupt	<input checked="" type="checkbox"/>	0	0	
TIM1 trigger and commutation interrupts	<input type="checkbox"/>	0	0	
TIM1 capture compare interrupt	<input type="checkbox"/>	0	0	

3.3.2. Gestione IMU

Successivamente è stata abilitata l'interfaccia I2C per consentire la comunicazione con l'IMU. La connessione è stata realizzata utilizzando due linee GPIO, configurate rispettivamente come SDA (Serial Data) e SCL (Serial Clock), collegate direttamente ai pin dedicati dell'IMU..

I2C1 Mode and Configuration				
Configuration				
<input type="button" value="Reset Configuration"/> <input checked="" type="checkbox"/> NVIC Settings <input checked="" type="checkbox"/> DMA Settings <input checked="" type="checkbox"/> GPIO Settings				
<input checked="" type="checkbox"/> Parameter Settings <input checked="" type="checkbox"/> User Constants				
Configure the below parameters :				
<input type="text" value="Search (Ctrl+F)"/> <input type="button" value="<"/> <input type="button" value=">"/>		i		
Timing configuration				
Custom Timing		Disabled		
I2C Speed Mode		Fast Mode		
I2C Speed Frequency (KHz)		400		
Rise Time (ns)		0		
Fall Time (ns)		0		
Coefficient of Digital Filter		0		
Analog Filter		Enabled		
Timing		0x00300F38		

In modalità Fast, il protocollo I2C opera a una frequenza di 400 kHz. Per acquisire i dati dall'IMU a una frequenza regolare di 100 Hz, è stato utilizzato un meccanismo asincrono basato su interrupt:

TIM1 è configurato per generare un interrupt ogni 10 ms. All'interno della relativa funzione di callback viene settata la flag ***flagTim1***, verificata poi nel ciclo principale (while(1)). Questo approccio, che combina l'uso di interrupt per la temporizzazione e polling per il controllo del flusso, può essere descritto come una **gestione asincrona tramite polling su flag**. In questo modo, l'acquisizione dell'orientamento del drone da parte dell'IMU avviene in modo regolare e non blocca l'esecuzione del programma principale.

Nella figura seguente è mostrata la configurazione dei pin utilizzati per la comunicazione I2C:

The screenshot shows the 'I2C1 Mode and Configuration' window. At the top, there's a 'Mode' section with tabs for Cortex-M7, Cortex-M4, and PowerDomain, with Cortex-M4 selected. Below that is an 'I2C' tab. The main area is titled 'Configuration' and contains a 'Reset Configuration' button and several checkboxes for 'Parameter Settings', 'User Constants', 'NVIC Settings', 'DMA Settings', and 'GPIO Settings', all of which are checked. A 'Search Signals' input field with placeholder text 'Search (Ctrl+F)' is present. A checkbox for 'Show only Modified Pins' is also visible. At the bottom, a table lists pin configurations:

Pin N...	Signal on...	Pin Conte...	GPIO ou...	GPIO mo...	GPIO Pu...	Maximum...	Fast Mode	User Label	Modified
PB6	I2C1_SCL	ARM Cor...	n/a	Alternate...	No pull-u...	Low	n/a		<input checked="" type="checkbox"/>
PB7	I2C1_SDA	ARM Cor...	n/a	Alternate...	No pull-u...	Low	n/a		<input checked="" type="checkbox"/>

Nelle seguenti immagini sono illustrate alcune funzioni implementate nel file *IMU.c*, utilizzate per la gestione della comunicazione e l'acquisizione dei dati dall'IMU **BNO055**:

```
1 #include "IMU.h"
2 #include <stdio.h>
3
4 #define ACC_LSB_TO_MS2      (1.0 / 100.0)
5 #define GYR_LSB_TO_DPS      (1.0 / 16.0)
6 #define MAG_LSB_TO_UT       (1.0 / 16.0)
7
8 I2C_HandleTypeDef *_bno055_i2c_port;
9
10 float magScale = 16;
11 float accelScale = 100;
12 float angularRateScale = 16;
13 float eulerScale = 16;
14 float quaScale = (1<<14);
15
16 void bno055_assignI2C(I2C_HandleTypeDef *hi2c_device)
17 {
18     _bno055_i2c_port = hi2c_device;
19 }
20
21 void bno055_delay(uint16_t ms)
22 {
23     HAL_Delay(ms);
24 }
25
26 void bno055_readData(uint8_t reg, uint8_t *buffer, uint8_t length)
27 {
28     uint16_t dev_addr = BN0055_I2C_ADDR << 1;
29
30     HAL_StatusTypeDef status = HAL_I2C_Mem_Read
31     (
32         _bno055_i2c_port,
33         dev_addr,
34         reg,
35         I2C_MEMADD_SIZE_8BIT,
36         buffer,
37         length,
38         HAL_MAX_DELAY
39     );
40
41     if (status != HAL_OK) {
42         printf("Errore nella lettura del registro 0x%02X\n", reg);
43     }
44 }
45
```

```

46 void bno055_writeData(uint8_t reg, uint8_t value)
47 {
48     uint16_t dev_addr = BN055_I2C_ADDR << 1;
49
50     HAL_StatusTypeDef status = HAL_I2C_Mem_Write
51     (
52         _bno055_i2c_port,
53         dev_addr,
54         reg,
55         I2C_MEMADD_SIZE_8BIT,
56         &value,
57         1,
58         HAL_MAX_DELAY
59     );
60
61     if (status != HAL_OK) {
62         printf("Errore nella scrittura del registro 0x%02X\n", reg);
63     }
64 }
65
66 void bno055_setPage(uint8_t page)
67 {
68     bno055_writeData(BN055_PAGE_ID, page);
69 }
70
71 void bno055_reset(void)
72 {
73     bno055_writeData(BN055_SYS_TRIGGER, BN055_VECTOR_QUATERNION);
74     bno055_delay(650);
75 }
76
77 void bno055_setOperationMode(bno055_opmode_t mode)
78 {
79     bno055_writeData(BN055_OPR_MODE, mode);
80     if (mode == BN055_OPERATION_MODE_CONFIG)
81     {
82         bno055_delay(19);
83     }
84     else
85     {
86         bno055_delay(7);
87     }
88 }
89
90 void bno055_setOperationModeConfig() {
91     bno055_setOperationMode(BN055_OPERATION_MODE_CONFIG);
92 }
93
94 void bno055_setOperationModeNDOF() {
95     bno055_setOperationMode(BN055_OPERATION_MODE_NDOF);
96 }
97
98 void bno055_init()
99 {
100     bno055_reset();
101
102     uint8_t id = 0;
103     bno055_readData(BN055_CHIP_ID, &id, 1);
104     if (id != BN055_CHIP_ID) {
105         printf("Can't find BN055, id: 0x%02x. Please check your wiring.\r\n", id);
106     }
107
108     bno055_setPage(0);
109     bno055_writeData(BN055_SYS_TRIGGER, 0x0);
110
111     bno055_setOperationModeConfig();
112     bno055_delay(10);
113 }
114

```

```

115 bno055_vector_t bno055_getVector(uint8_t vec)
116 {
117     bno055_setPage(0);
118     uint8_t buffer[8];
119
120     if (vec == BN0055_VECTOR_QUATERNION)
121     {
122         bno055_readData(vec, buffer, 8);
123     } else
124     {
125         bno055_readData(vec, buffer, 6);
126     }
127
128     double scale = 1;
129
130     if (vec == BN0055_MAG_DATA_X_LSB)
131     {
132         scale = magScale;
133     } else if (vec == BN0055_ACC_DATA_X_LSB ||
134                vec == BN0055_LIA_DATA_X_LSB ||
135                vec == BN0055_VECTOR_GRAVITY) {
136         scale = accelScale;
137     } else if (vec == BN0055_GYR_DATA_X_LSB) {
138         scale = angularRateScale;
139     } else if (vec == BN0055_VECTOR_EULER) {
140         scale = eulerScale;
141     } else if (vec == BN0055_VECTOR_QUATERNION) {
142         scale = quaScale;
143     }
144
145     bno055_vector_t xyz = {.w = 0, .x = 0, .y = 0, .z = 0};
146
147     xyz.x = ((int16_t)((buffer[1] << 8) | buffer[0])) / scale;
148     xyz.y = ((int16_t)((buffer[3] << 8) | buffer[2])) / scale;
149     xyz.z = ((int16_t)((buffer[5] << 8) | buffer[4])) / scale;
150
151     if (vec == BN0055_VECTOR_QUATERNION)
152     {
153         xyz.w = ((int16_t)((buffer[7] << 8) | buffer[6])) / scale;
154     }
155
156     return xyz;
157 }
158
159 bno055_vector_t bno055_getVectorEuler()
160 {
161     return bno055_getVector(BN0055_VECTOR_EULER);
162 }
163

```

Per evitare il sovraccarico della comunicazione seriale, la stampa è limitata dalla variabile di flag *flag_print*, aggiornata ogni 10 periodi del Timer 1 tramite la sua callback. L'interrupt di TIM1 incrementa un contatore che, raggiunto il valore 10, aggiorna *flag_print* per consentire la stampa. In questo modo, la frequenza di aggiornamento dei dati seriali è ridotta a 100 ms, ossia dieci volte inferiore a quella del Timer 1 (10 ms):

```
● ● ●
810 void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
811 {
812     flagTim1 = 1; //Flag per far partire la prima volta in assoluto il codice nel while
813     if(htim == &htim1)
814     {
815         //Tutto questo codice serve per definire la velocità di stampa del printf che è 10 volte
816         //minore del periodo di TIM1
817         // periodo TIM1 = 10ms, periodo di stampa di 1 singolo carattere = 100ms
818         if(cont == 10)
819         {
820             cont = 0; //riazzera il contatore dopo aver stampato un singolo carattere nel while(1)
821             flag_print = 1;
822         }
823         else
824             cont++;
825     }
}
```

La trasmissione carattere per carattere tramite la periferica UART3 avviene attraverso la funzione generica *_io_putchar(int ch)*, che reindirizza l'output alla porta seriale utilizzando *HAL_UART_Transmit*. Questa funzione è fondamentale per implementare la stampa tramite *printf* in ambienti embedded, permettendo di inviare i dati a terminali o dispositivi via UART. Nel nostro caso, è stata cruciale per visualizzare i dati inizialmente su PuTTY e successivamente sulla seriale di MATLAB, facilitando la registrazione dei dati e la loro analisi tramite grafici.

3.3.3. PWM

La modulazione a larghezza d'impulso (PWM, Pulse Width Modulation) è una tecnica di controllo digitale utilizzata per regolare la potenza erogata a un carico elettrico variando il **duty cycle**, ovvero la percentuale di tempo in cui il segnale rimane nello stato alto durante un ciclo.

Nel nostro caso, i segnali PWM vengono generati utilizzando un timer configurato in modalità PWM, che produce impulsi con duty cycle variabili, regolabili in base ai parametri impostati.

Il duty cycle (δ) può essere calcolato con la formula:

$$\delta = \frac{T_{on}}{T} \cdot 100\%$$

- **T_{on}** è la durata dell'intervallo in cui il segnale è attivo (alto),
- **T** è il periodo totale del segnale.

Quando il timer è configurato in modalità PWM, ad ogni evento generato viene associato un valore di uscita PWM tramite un registro chiamato *CCRx* (Capture/Compare Register). Questo registro controlla la durata dell'impulso attivo e quindi il *duty cycle*, secondo la relazione:

$$CCRx = MaxCNT \cdot (1 - \delta)$$

- **CCRx** è il valore che rappresenta il *duty cycle*,
- **MaxCNT** è il valore massimo raggiungibile dal contatore del timer.

Poiché il valore di *CCRx* dipende dalla frequenza di clock effettiva del timer (*TIMxCLK*), la frequenza con cui il canale PWM viene aggiornato può essere calcolata con la formula:

$$CHxUpdate = \frac{TIMxCLK}{MaxCNT}$$

Questa formula definisce la frequenza alla quale viene generato il segnale PWM, che è fondamentale per il controllo preciso di dispositivi come motori, LED, o altri attuatori.

Di seguito sono riportate le immagini relative alla configurazione del PWM all'interno del file .ioc del progetto:

TIM3 Mode and Configuration

Mode		
Cortex-M7	Cortex-M4	PowerDomain
<input type="checkbox"/>	<input checked="" type="checkbox"/>	D2
Slave Mode	Disable	▼
Trigger Source	Disable	▼
Clock Source	Internal Clock	▼
Channel1	PWM Generation CH1	▼
Channel2	PWM Generation CH2	▼
Channel3	PWM Generation CH3	▼
Channel4	PWM Generation CH4	▼
Combined Channels	Disable	▼
<input type="checkbox"/> ETR IO as Clearing Source <input type="checkbox"/> XOR activation <input type="checkbox"/> One Pulse Mode		

Configuration

Reset Configuration	NVIC Settings	DMA Settings	GPIO Settings
Parameter Settings	Parameter Settings	User Constants	User Constants
Configure the below parameters :	<input type="checkbox"/> Search (Ctrl+F) <input type="checkbox"/> <input type="checkbox"/>		
▼ Counter Settings	Prescaler (PSC - 16 bits value) : 64-1 Counter Mode : Up Counter Period (AutoReload Re... : 20000-1 Internal Clock Division (CKD) : No Division auto-reload preload : Disable		
▼ Trigger Output (TRGO) Parameters	Master/Slave Mode (MSM bit) : Disable (Trigger input effect not delayed) Trigger Event Selection TRGO : Reset (UG bit from TIMx_EGR)		
▼ Clear Input	Clear Input Source : Disable		
▼ PWM Generation Channel 1	Mode : PWM mode 1		

Parameter Settings									
User Constants									
NVIC Settings									
DMA Settings									
GPIO Settings									
Search Signals									
<input type="text" value="Search (Ctrl+F)"/>									
<input type="checkbox"/> Show only Modified Pins									
Pin Na...	Signal on ...	Pin Conte...	GPIO out...	GPIO mode	GPIO Pul...	Maximum...	Fast Mode	User Label	Modified
PA6	TIM3_CH1	ARM Cort...	n/a	Alternate ...	No pull-u...	Low	n/a		<input checked="" type="checkbox"/>
PB1	TIM3_CH4	ARM Cort...	n/a	Alternate ...	No pull-u...	Low	n/a		<input checked="" type="checkbox"/>
PC7	TIM3_CH2	ARM Cort...	n/a	Alternate ...	No pull-u...	Low	n/a		<input checked="" type="checkbox"/>
PC8	TIM3_CH3	ARM Cort...	n/a	Alternate ...	No pull-u...	Low	n/a		<input checked="" type="checkbox"/>

Nel nostro progetto, il timer utilizzato è il **TIM3**, configurato con il clock di riferimento impostato su "Internal Clock". I quattro canali associati a questo timer sono stati configurati in modalità "**PWM Generation**", ovvero generazione di segnali a modulazione di larghezza d'impulso. I parametri fondamentali per la frequenza di aggiornamento del PWM sono il **Prescaler** e il **Counter Period**.

Nel nostro caso, assumendo un clock del timer pari a 64 MHz, un prescaler di 64 e un periodo di 20.000, otteniamo:

$$UpdateEvent = \frac{TimerClock}{(Prescaler + 1)(Period + 1)}$$

$$UpdateEvent = \frac{64000000 \text{ MHz}}{(64 + 1)(20000 + 1)} = 50\text{Hz} = 20\text{ms}$$

Questa frequenza è tipica per il controllo degli ESC usati nei motori brushless, come quelli impiegati nei droni.

Passando alla sezione **GPIO Settings**, possiamo osservare la configurazione dei quattro pin a cui sono collegati i segnali PWM. Ogni pin è collegato a un ESC, a sua volta connesso a un motore. Questa configurazione hardware è cruciale, in quanto consente di inviare un segnale PWM personalizzato a ciascun motore. In questo modo, il software può regolare finemente la velocità di rotazione di ogni elica, garantendo il **bilanciamento dinamico** e la stabilità del volo.

Nella prossima sezione sarà analizzata la parte di codice che gestisce la generazione dei segnali PWM, facendo riferimento alle funzioni contenute nel file ‘map.c’

```

38 float calcoloDuty(float speed)
39 {
40     float duty = (((MAX_DUTY - MIN_DUTY)*speed) + ((MIN_DUTY * MAX_SPEED)-(MAX_DUTY *
41         MIN_SPEED)))/(MAX_SPEED - MIN_SPEED);
42     if (duty < MIN_DUTY) duty = MIN_DUTY;
43     else if (duty > MAX_DUTY) duty = MAX_DUTY;
44     return duty;
45 }
46

```

La funzione *calcoloduty(float speed)*, prende i valori delle velocità per mapparli in duty tramite la formula:

$$\text{duty} = \frac{(\text{MAX}_{\text{DUTY}} - \text{MIN}_{\text{DUTY}}) \cdot \text{speed}}{\text{MAX}_{\text{SPEED}} - \text{MIN}_{\text{SPEED}}} + \frac{(\text{MIN}_{\text{DUTY}} \cdot \text{MAX}_{\text{SPEED}}) - (\text{MAX}_{\text{DUTY}} \cdot \text{MIN}_{\text{SPEED}})}{\text{MAX}_{\text{SPEED}} - \text{MIN}_{\text{SPEED}}}$$

che rappresenta la **relazione lineare duty e velocità**. Questo valore verrà poi utilizzato per convertire i duty in PWM tramite la funzione *setPwm* che analizziamo di seguito:

```

5 //Funzione che dati i duty cycle per ogni canale del timer associa i 4 rispettivi valori ai registri
6 //CCR per creare i segnali PWM
6 void setPwm(float pwm1, float pwm2, float pwm3, float pwm4)
7 {
8     //pwmX è un duty cycle in percentuale
9     pwm1 = pwm1 > MAX_DUTY ? MAX_DUTY : pwm1 < MIN_DUTY ? MIN_DUTY : pwm1; //Condizione con operatore
10    ternario
11    pwm2 = pwm2 > MAX_DUTY ? MAX_DUTY : pwm2 < MIN_DUTY ? MIN_DUTY : pwm2;
12    pwm3 = pwm3 > MAX_DUTY ? MAX_DUTY : pwm3 < MIN_DUTY ? MIN_DUTY : pwm3;
13    pwm4 = pwm4 > MAX_DUTY ? MAX_DUTY : pwm4 < MIN_DUTY ? MIN_DUTY : pwm4;
14    TIM3->CCR1 = (uint32_t)(TIM3->ARR * pwm1/100);
15    TIM3->CCR2 = (uint32_t)(TIM3->ARR * pwm2/100);
16    TIM3->CCR3 = (uint32_t)(TIM3->ARR * pwm3/100);
17    TIM3->CCR4 = (uint32_t)(TIM3->ARR * pwm4/100);
18 }

```

Tale funzione ci permette di assegnare i valori di duty agli appositi registri rimanendo in un range di sicurezza. Entrambe le funzioni vengono utilizzate nel file ‘main.c’, come mostrato in sezione “*Funzionamento complessivo*”.

3.3.4. Radiocomando

Per il radiocomando ci siamo serviti di due timer, il TIM5 e il TIM2, configurati entrambi con gli stessi parametri che vengono elencati di seguito.

TIM5 Mode and Configuration

Mode

Cortex-M7	Cortex-M4	PowerDomain D2
<input type="checkbox"/>	<input checked="" type="checkbox"/>	

Runtime contexts:

- Slave Mode: Disable
- Trigger Source: Disable
- Clock Source: Internal Clock
- Channel1: Disable
- Channel2: Disable
- Channel3: Disable
- Channel4: Disable
- Combined Channels: PWM Input on CH1

ETR IO as Clearing Source

XOR activation

One Pulse Mode

TIM5 Mode and Configuration

Configuration

Reset Configuration

Parameter Settings **User Constants** **NVIC Settings** **DMA Settings** **GPIO Settings**

Configure the below parameters :

Search (Ctrl+F)

Counter Settings

- Prescaler (PSC - 16 bits value): 64-1
- Counter Mode: Up
- Counter Period (AutoReload Register - 32 bits va...): 20000-1
- Internal Clock Division (CKD): No Division
- auto-reload preload: Disable

Trigger Output (TRGO) Parameters

- Master/Slave Mode (MSM bit): Disable (Trigger input effect not delayed)
- Trigger Event Selection TRGO: Reset (UG bit from TIMx_EGR)

PWM Input CH1

- Input Trigger: TI1FP1
- Slave Mode Controller: Reset Mode
- Parameters for Channel 1
- Polarity Selection: Rising Edge

IC Selection

- Prescaler Division Ratio: Direct
- Input Filter (4 bits value): No division
- Parameters for Channel 2
- Polarity Selection (opposite CH1): Falling Edge
- IC Selection: Indirect
- Prescaler Division Ratio: No division
- Input Filter (4 bits value): 0

Parameter Settings **User Constants** **NVIC Settings** **DMA Settings** **GPIO Settings**

Search Signals

Search (Ctrl+F)

Show only Modified Pins

Pin Name	Signal on	Pin Context	GPIO output	GPIO mode	GPIO Pull-up	Maximum	Fast Mode	User Label	Modified
PA0	TIM5_CH1	ARM Cortex... n/a	n/a	Alternate ...	No pull-u...	Low	n/a		<input checked="" type="checkbox"/>

Il TIM5 CH1, collegato al pin PA0, riceve un segnale PWM dal channel 5 del ricevitore. Misurando la durata dell'impulso tra il fronte di salita e quello di discesa, è possibile determinare lo stato dell'interruttore e quindi gestire le transizioni tra spegnimento, armamento ed esecuzione dei motori.

Di seguito vengono mostrati i settaggi del TIM2:

TIM2 Mode and Configuration

Mode

Runtime contexts:

Cortex-M7	Cortex-M4	PowerDomain
<input type="checkbox"/>	<input checked="" type="checkbox"/>	D2

Slave Mode: Disable

Trigger Source: Disable

Clock Source: Internal Clock

Channel1: Disable

Channel2: Disable

Channel3: Disable

Channel4: Disable

Combined Channels: PWM Input on CH1

Use ETR as Clearing Source: Disable

XOR activation

One Pulse Mode

Configuration

Reset Configuration

Configure the below parameters :

Parameter Settings | User Constants | NVIC Settings | DMA Settings | GPIO Settings

Counter Settings

- Prescaler (PSC - 16 bits value) : 64-1
- Counter Mode : Up
- Counter Period (AutoReload Register - 32 bits va... 20000-1
- Internal Clock Division (CKD) : No Division
- auto-reload preload : Disable

Trigger Output (TRGO) Parameters

- Master/Slave Mode (MSM bit) : Disable (Trigger input effect not delayed)
- Trigger Event Selection TRGO : Reset (UG bit from TIMx_EGR)

PWM Input CH1

- Input Trigger : TI1FP1
- Slave Mode Controller : Reset Mode
- Parameters for Channel 1
- Polarity Selection : Rising Edge

PWM Input CH2

- IC Selection : Direct
- Prescaler Division Ratio : No division
- Input Filter (4 bits value) : 0
- Parameters for Channel 2
- Polarity Selection (opposite CH1) : Falling Edge
- IC Selection : Indirect
- Prescaler Division Ratio : No division
- Input Filter (4 bits value) : 0

Parameter Settings | User Constants | NVIC Settings | DMA Settings | GPIO Settings

Search Signals

Search (Ctrl+F)

Show only Modified Pins

Pin Na...	Signal on ...	Pin Conte...	GPIO out...	GPIO mode	GPIO Pul...	Maximum...	Fast Mode	User Label	Modified
PA5	TIM2_CH1	ARM Cortex...	n/a	Alternate ...	No pull-u...	Low	n/a		<input checked="" type="checkbox"/>

Il TIM2 CH1, collegato al pin PA5, riceve invece il segnale PWM dal channel 3 del radiocomando. Anche in questo caso, la misura della durata dell’impulso consente di interpretare la posizione della levetta sinistra e aggiornare il riferimento di yaw per il relativo controllore PID. Questo controllo è attivo solo quando il motore è in esecuzione (interruttore su channel 5 in posizione alta).

Infine, vengono mostrati alcuni snapshot del file '**main.c**' relativi alla gestione del radiocomando.

```

1 //Radiocomando
2 void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim)
3 {
4     if(htim == &htim5)
5     {
6         if (htim->Channel == HAL_TIM_ACTIVE_CHANNEL_1) // If the interrupt is triggered by channel 1
7         {
8             // Read the IC value
9             uint32_t ICValue = HAL_TIM_ReadCapturedValue(htim, TIM_CHANNEL_1);
10
11            if (ICValue != 0)
12            {
13                // calculate the Duty Cycle
14                uint32_t duty_received = (HAL_TIM_ReadCapturedValue(htim, TIM_CHANNEL_2)
15                *100)/ICValue;
16
17                if(duty_received >= 11)
18                    i = 0;
19                else if (duty_received <= 8)
20                    i = 2;
21                else
22                    i = 1;
23            }
24        }
25
26
27        if(htim == &htim2)
28        {
29            if (htim->Channel == HAL_TIM_ACTIVE_CHANNEL_1) // If the interrupt is triggered by channel 1
30            {
31                // Read the IC value
32                uint32_t ICValue = HAL_TIM_ReadCapturedValue(htim, TIM_CHANNEL_1);
33
34                if (ICValue != 0)
35                {
36                    // calculate the Duty Cycle
37                    uint32_t duty_received = (HAL_TIM_ReadCapturedValue(htim, TIM_CHANNEL_2)
38                    *100)/ICValue;
39
40                    if(duty_received >= 14)
41                    {
42                        rif_yaw = 1;
43                        yaw_angolo = 20; //In sostituzione a rif_yaw
44                    }
45                    else if (duty_received <= 7)
46                    {
47                        rif_yaw = -1;
48                        yaw_angolo = -20;
49                    }
50                    else
51                    {
52                        rif_yaw = 0;
53                        yaw_angolo = 0;
54                    }
55                }
56            }
57        }
58    }
}

```

NOTA BENE: La mappatura programmata è valida solo se il duty cycle ricevuto dal combined PWM rispetta specifici intervalli.

Calcolo fasce di duty con Oscilloscopio

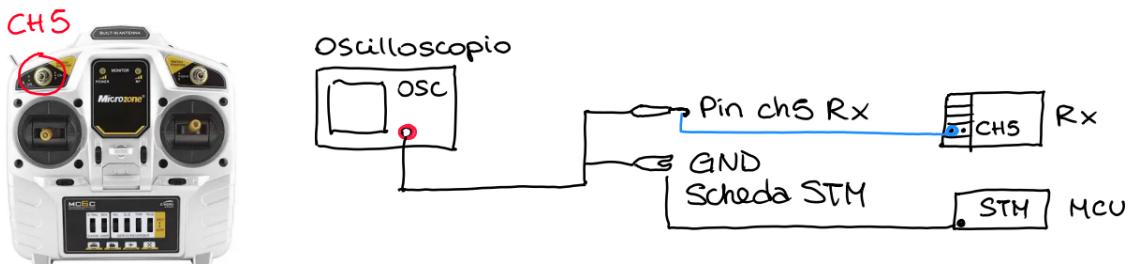
Calcolo del duty inviato dal Trasmettitore del radiocomando al Ricevitore sulla scheda, in base alla posizione delle due leve utilizzate.

Utilizzo dell'oscilloscopio per visualizzare le funzioni d'onda e trovarne il periodo totale (T) e quello di salita (T_{on}), per poi calcolare il duty.

Collegamento fisico:

- Cavo per l'oscilloscopio: un'estremità collegata all'oscilloscopio e l'altra con 2 attacchi: uno ad uncino e uno con un morsetto.
- Attacco ad uncino collegato al pin del canale del ricevitore.
- Morsetto connesso ad un GND qualsiasi della scheda STM32.

CHANNEL 5

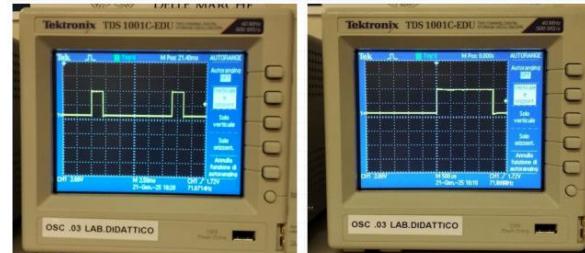


Leva giù (motori spenti)

$$T = 14 \text{ ms}$$

$$T_{\text{on}} = 2 \text{ ms}$$

$$\text{duty (\%)} = (T_{\text{on}}/T) 100 = 14,29\%$$

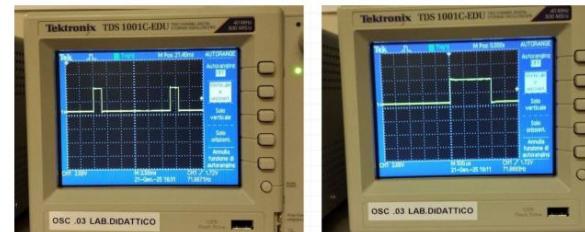


Leva al centro (armamento)

$$T = 14 \text{ ms}$$

$$T_{\text{on}} = 1,5 \text{ ms}$$

$$\text{duty (\%)} = 10,71\%$$

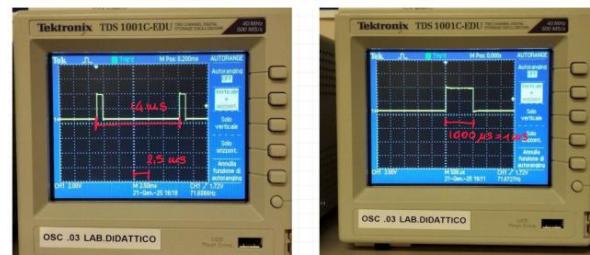


Leva su (esecuzione)

$$T = 14 \text{ ms}$$

$$T_{\text{on}} = 1 \text{ ms}$$

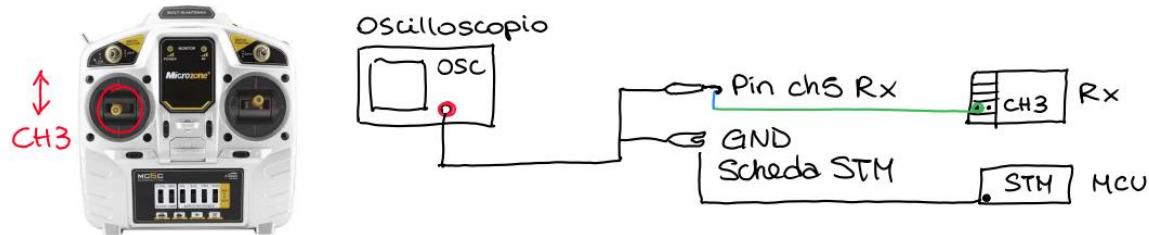
$$\text{duty (\%)} = 7,14\%$$



Ricollegando tali valori del duty alle condizioni di stato dei motori del droni, si ha che il comportamento dei motori varia secondo le seguenti fasce di duty del segnale inviato al Ricevitore:

- $\text{duty_received} < 7 \rightarrow$ esecuzione
- $7 < \text{duty_received} < 11 \rightarrow$ armamento
- $\text{duty_received} > 11 \rightarrow$ motori spenti

CHANNEL 3



I valori di duty dei segnali PWM sono gli stessi relativi al channel 5.

Approssimando i valori si ha:

Leva completamente giù

duty ~ 14,5%

Leva perfettamente al centro

duty ~ 11%

Leva completamente su

duty ~ 8%

Per discretizzare i valori continui di questa leva e regolare il movimento di imbardata del drone, abbiamo suddiviso gli stati come segue:

- $\text{duty_received} \geq 14 \rightarrow \text{rif_yaw} = 1$ (imbardata di 20° in senso orario)
- $\text{duty_received} \leq 7 \rightarrow \text{rif_yaw} = -1$ (imbardata di 20° in senso antiorario)
- $7 \leq \text{duty_received} \leq 14 \rightarrow \text{rif_yaw} = 0$ (nessuna imbardata)

Quindi:

1) Il segnale ricevuto tramite **TIM5 CH1** (PA0) viene analizzato calcolando il duty cycle, e in base al suo valore vengono distinti tre stati:

- Se il duty cycle è **maggiore o uguale all'11%**, viene selezionato uno stato ($i = 0$).
- Se il duty cycle è **inferiore o uguale all'8%**, viene selezionato un altro stato ($i = 2$).
- Altrimenti viene assegnato lo stato intermedio ($i = 1$).

2) Analogamente, il segnale ricevuto tramite **TIM2 CH1** (PA5) viene elaborato per aggiornare i riferimenti di yaw:

- Se il duty cycle è **maggiore o uguale al 14%**, viene impostato $rif_yaw = 1$ con un angolo di rotazione $+20^\circ$.
- Se il duty cycle è **minore o uguale al 7%**, viene impostato $rif_yaw = -1$ con un angolo di rotazione -20° .
- In tutti gli altri casi, rif_yaw viene azzerato e non c'è nessuna rotazione.

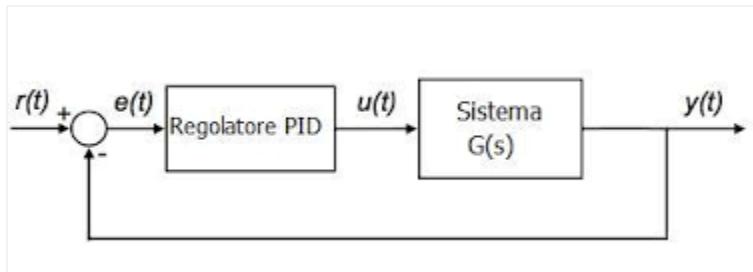
È possibile correggere finemente il valore del duty cycle utilizzando i **tasti di precisione** ("Elevator Trim" e "Aileron fine-tuning") presenti nella parte destra del radiocomando. Questi tasti agiscono rispettivamente sui segnali che influenzano il controllo di pitch e roll, permettendo così di centrare il duty cycle entro le soglie corrette per un funzionamento stabile.

3.3.5. PID

I **controllori PID** (Proporzionale, Integrale, Derivativo) sono algoritmi progettati per regolare sistemi in **anello chiuso**, dove il controllo si basa su un meccanismo di **feedback**. L'obiettivo è mantenere una variabile del sistema (variabile di processo) il più possibile vicina a un valore desiderato (setpoint), correggendo l'errore tra i due.

La variabile di controllo $u(t)$ è ottenuta combinando tre termini:

- **proporzionale**: reagisce in proporzione all'errore istantaneo,
- **integrale**: tiene conto dell'accumulo dell'errore nel tempo,
- **derivativo**: anticipa l'andamento dell'errore osservandone la variazione.



Il controllore PID standard si esprime nella forma:

- $r(t)$: segnale di riferimento in ingresso;
- $y(t)$: segnale di uscita del sistema di controllo in retroazione;
- $e(t)$: errore dovuto alla differenza algebrica tra il segnale di riferimento in ingresso $r(t)$ e l'uscita $y(t)$;
- $u(t)$: ingresso di controllo.

Nel classico schema a retroazione unitaria, il controllore viene posto in serie al sistema da controllare (prima di esso), così da agire direttamente sulla grandezza in ingresso.

Il circuito di retroazione riporta l'uscita $y(t)$ all'ingresso, tramite un nodo sommatore che calcola l'errore $e(t)$. Questa configurazione viene detta **struttura standard**.

La legge di controllo implementata dal regolatore PID è la seguente:

$$u(t) = K_P \cdot e(t) + K_I \cdot \int_{t_0}^t e(\tau) d\tau + K_D \frac{de(t)}{dt}$$

dove:

- K_P è il guadagno proporzionale,
- K_I è il guadagno integrale,
- K_D è il guadagno derivativo.

Una formulazione alternativa, molto usata, evidenzia le **costanti di tempo**:

$$u(t) = K_P \cdot \left(e(t) + \frac{1}{T_I} \cdot \int_0^t e(\tau) d\tau + T_D \frac{de(t)}{dt} \right)$$

dove:

- T_I è la costante di tempo dell'azione integrativa,
- T_D è la costante di tempo dell'azione derivativa.

I parametri che caratterizzano il PID - K_P , T_I (o K_I), e T_D (o K_D) - sono detti anche **gradi di libertà** del controllore.

Una scelta errata di questi valori può portare a instabilità del sistema controllato, con uscita divergente e/o oscillazioni incontrollate.

Nella nostra implementazione, eseguendo i calcoli seguendo la struttura del codice sviluppato, siamo risaliti ai valori ottimali dei parametri PID, al fine di garantire la stabilità e le prestazioni desiderate del sistema:

$$K_{PR} = 0.016, \quad K_{PP} = 0.016, \quad K_{PY} = 0.04$$

$$K_{IR} = 0.0001, \quad K_{IP} = 0.0002, \quad K_{IY} = 0$$

$$K_{DR} = 0.06, \quad K_{DP} = 0.06, \quad K_{DY} = 0$$

L'azione proporzionale inizialmente impostata non era sufficiente a garantire la stabilità del sistema, poiché induceva oscillazioni attorno al valore di riferimento.

Per migliorare il comportamento dinamico, è stato necessario introdurre un termine derivativo che ha consentito di smorzare le oscillazioni e stabilizzare la risposta del sistema.

Successivamente, è stata aggiunta anche una componente integrativa di entità molto ridotta, con l'obiettivo di correggere lentamente eventuali errori statici senza compromettere la stabilità complessiva.

Nella sezione dedicata ai test nel documento sono presentate **immagini e descrizioni** degli esperimenti effettuati per la calibrazione.

Infine, nel file **PID.c**, troviamo le funzioni dedicate all'implementazione del controllore PID, comprese quelle che:

- inizializzano la struttura PID con i relativi coefficienti;
- calcolano il valore della regolazione in tempo reale a partire dall'errore misurato.

```
10 //Assegnamo i valori ai coefficienti della struct del PID
11 // Init e Tune PID
12 void PID_Init(PID* pid, float kp, float ki, float kd, float dt, float outMin, float outMax)
13 {
14     pid->kp = kp;
15     pid->ki = ki;
16     pid->kd = kd;
17     pid->dt = dt;
18     pid->Iterm = 0;
19     pid->DtermFiltered = 0;
20     pid->lastError = 0;
21     pid->outMax = outMax;
22     pid->outMin = outMin;
23 }
```

Ora riportiamo la funzione che calcola l'uscita del PID conoscendo l'errore in tempo reale.

```
● ● ●  
27 float PID_Controller(PID* pid, float input, float setPoint)  
28 {  
29     float output;  
30     float newIterm;  
31  
32     float error = setPoint - input;  
33  
34     float Pterm = pid->kp * error;  
35  
36     newIterm = pid->Iterm + (pid->ki) * pid->dt * pid->lastError;  
37  
38     float Dterm = (pid->kd/pid->dt) * (error - pid->lastError);  
39  
40     pid->lastError = error;  
41  
42     output = Pterm + newIterm + Dterm;  
43  
44     // Saturazione dell'output & anti-windup  
45     if (output > pid->outMax) {  
46         output = pid->outMax;  
47     } else if (output < pid->outMin) {  
48         output = pid->outMin;  
49     } else {  
50         pid->Iterm = newIterm; // Applica anti-windup  
51     }  
52  
53     return output;  
54 }
```

Nella funzione successiva, ad ogni elemento dell'array **Speeds_quad[]** viene assegnato un valore che risulta dalle somme ottenute applicando la matrice di controllo.

Viene poi calcolata la radice quadrata dei valori appena ottenuti, in modo da ottenere le velocità **Speeds[]** che corrispondono alle velocità reali dei motori, senza produrre valori negativi:

```
70 float* SpeedCompute(float virtualInputs[])
71 {
72     static float Speeds_quad[4];
73     static float Speeds[4];
74
75     Speeds_quad[0] = (1/(4*b))*virtualInputs[0] - (1/(2*l*b))*virtualInputs[2] +
    (1/(4*d))*virtualInputs[3];
76     Speeds_quad[1] = (1/(4*b))*virtualInputs[0] - (1/(2*l*b))*virtualInputs[1] -
    (1/(4*d))*virtualInputs[3];
77     Speeds_quad[2] = (1/(4*b))*virtualInputs[0] + (1/(2*l*b))*virtualInputs[2] +
    (1/(4*d))*virtualInputs[3];
78     Speeds_quad[3] = (1/(4*b))*virtualInputs[0] + (1/(2*l*b))*virtualInputs[1] -
    (1/(4*d))*virtualInputs[3];
79
80     /*
81      * Calcoliamo le velocità dei motori al quadrato, poiché non possono essere negative.
82      * Partendo dal valore di throttle e seguendo le matrici di controllo dei droni,
83      * andiamo a sommare e sottrarre le variabili date tramite il PID per il controllo delle velocità.
84      */
85
86     Speeds[0] = sqrt(Speeds_quad[0]);
87     Speeds[1] = sqrt(Speeds_quad[1]);
88     Speeds[2] = sqrt(Speeds_quad[2]);
89     Speeds[3] = sqrt(Speeds_quad[3]);
90
91     // Una volta calcolata la velocità dei motori al quadrato, viene eseguita la radice
92
93     return Speeds;
94 }
```

3.4. Funzionamento complessivo

3.4.1. Inizializzazione delle periferiche

Nel file **main.c**, viene eseguita la prima parte dell'inizializzazione, che include tutte le strutture necessarie per il corretto funzionamento del sistema.

```
/* Initialize all configured peripherals */
MX_GPIO_Init();
MX_TIM1_Init();
MX_TIM3_Init();
MX_TIM2_Init();
MX_TIM5_Init();
MX_USART3_UART_Init();
MX_I2C1_Init();
/* USER CODE BEGIN 2 */

HAL_TIM_Base_Start_IT(&htim1);

HAL_TIM_IC_Start_IT(&htim2, TIM_CHANNEL_1);
HAL_TIM_IC_Start(&htim2, TIM_CHANNEL_2);

HAL_TIM_PWM_Start(&htim3, TIM_CHANNEL_1);
HAL_TIM_PWM_Start(&htim3, TIM_CHANNEL_2);
HAL_TIM_PWM_Start(&htim3, TIM_CHANNEL_3);
HAL_TIM_PWM_Start(&htim3, TIM_CHANNEL_4);

HAL_TIM_IC_Start_IT(&htim5, TIM_CHANNEL_1);
HAL_TIM_IC_Start(&htim5, TIM_CHANNEL_2);

bno055_assignI2C(&hi2c1);
bno055_init();
bno055_setOperationModeNDOF();

PID_Init(&PitchPID, kpp, kip, kdp, dt, -1.3, 1.3);
PID_Init(&RollPID, kpr, kir, kdr, dt, -1.3, 1.3);
PID_Init(&YawPID, kpy, kiy, kpy, dt, -1.3, 1.3); //Valori di prova

/* USER CODE END 2 */
```

Prima di tutto, vengono **inizializzati i GPIO** necessari per l'uso dei LED sulla scheda. Successivamente, vengono configurati i **timer** (TIM1, TIM2, TIM3, TIM5) per gestire temporizzazioni, intercettazioni di segnali e generazione di PWM. Viene anche configurato l'**I2C** per la comunicazione con l'IMU e l'UART per la comunicazione seriale.

Nel blocco di codice sotto "USER CODE BEGIN 2", vengono eseguite ulteriori inizializzazioni, tra cui:

- Avvio di tutti i timer, con le relative modalità di funzionamento e update.
- Configurazione dell'IMU: il dispositivo viene inizializzato, a cui vengono assegnati i parametri per l'operazione in modalità NDOF (Nine Degrees of Freedom).
- Inizializzazione delle 3 strutture PID di roll, pitch e yaw con i relativi coefficienti.

3.4.2. Funzionamento del Codice all'interno del while

Il codice all'interno del while principale è progettato per gestire i vari stati del sistema in base al valore della variabile i. La variabile i può assumere tre valori distinti, ognuno dei quali rappresenta uno stato specifico del drone:

1. **MOTORI FERMI (Stato 0, Default)**
2. **ARMATURA DEI MOTORI (Stato 1)**
3. **ESECUZIONE (Stato 2)**

L'immagine del codice che rappresenta questa logica è riportata nella pagina successiva:

```

/* USER CODE BEGIN WHILE */
while (1)
{
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */

    if(flagTim1)
    {
        if(calibrateFlag)
        {
            calibrateFlag = 0;
            ESC_Calibrate(); //ATTACCA LA BATTERIA DOPO AVER SETTATO IL PWM PER GLI ESC = LIMIT_DUTY
        }

        if(final)
        {
            switch (i) //Vedere come mettere più istruzioni in un singolo case dello switch
            {
                case 0: // Stato 0: Motore spento
                    stopMotors();
                    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0, GPIO_PIN_RESET); //Led verde
                    HAL_GPIO_WritePin(GPIOE, GPIO_PIN_1, GPIO_PIN_RESET); //Led giallo
                    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_14, GPIO_PIN_SET); //Led rosso
                    break;

                case 1: // Stato 1: Motore in armamento
                    armMotors();
                    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0, GPIO_PIN_RESET); //Led verde
                    HAL_GPIO_WritePin(GPIOE, GPIO_PIN_1, GPIO_PIN_SET); //Led giallo
                    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_14, GPIO_PIN_RESET); //Led rosso
                    break;

                case 2: // Stato 2: Motore in esecuzione
                    stabilizeMotors();

                    /*
                    if(calcoloDutyToSpeedFlag)
                    {
                        calcoloDutyToSpeedFlag = 0;
                        dutyMinYaw = findDutyYaw();
                    }
                    if(flag_print)
                    {
                        printf("Speed13: %.2f\r\n", dutyMinYaw);
                        flag_print=0;
                    }
                    */

                    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0, GPIO_PIN_SET); //Led verde
                    HAL_GPIO_WritePin(GPIOE, GPIO_PIN_1, GPIO_PIN_RESET); //Led giallo
                    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_14, GPIO_PIN_RESET); //Led rosso
                    break;

                default: // Motori fermi
                    stopMotors();
                    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0, GPIO_PIN_RESET); //Led verde
                    HAL_GPIO_WritePin(GPIOE, GPIO_PIN_1, GPIO_PIN_RESET); //Led giallo
                    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_14, GPIO_PIN_SET); //Led rosso
                    break;
            }
        }
    }
}
/* USER CODE END 3 */

```

La gestione di questi stati avviene all'interno di uno switch che verifica il valore della “i” ed esegue le operazioni corrispondenti a ciascun stato. Ogni stato è associato a una serie di operazioni che coinvolgono il controllo dei motori e il feedback visivo tramite i LED.

Stato 0 (i = 0, Default) Motori Fermi

All'inizio, la variabile i è impostata su 0 (LEVA GIU), il che indica che i motori sono spenti e il sistema è fermo. Le seguenti operazioni vengono eseguite:

- **Funzione:** stopMotors(), che ferma i motori tramite la funzione setPwmNoControl() (vedi più sotto).
- **LEDs:** Il LED rosso si accende per segnalare che i motori sono disattivati, mentre i LED verde e giallo sono spenti.

Stato 1 (i = 1) Armamento dei Motori

Quando la variabile i viene impostata a 1 (LEVA CENTRO). In questa fase vengono eseguite le seguenti operazioni:

- **Funzione:** armMotors(), che prepara i motori all'avvio regolando i PWM per i motori.
- **LEDs:** Viene acceso il LED giallo per segnalare che i motori sono in fase di armamento. Il LED rosso si spegne, ma il LED verde rimane spento durante questa fase.

Stato 2 (i = 2) Esecuzione

Quando la variabile i è impostata a 2 (LEVA SU), il drone entra nella fase di esecuzione attiva. In questa fase, il sistema esegue il controllo dei motori in base ai dati sensoriali e al controllo PID per mantenere la stabilità. Vengono eseguite tali operazioni:

- 1) **Funzione:** stabilizeMotors(), che regola la velocità dei motori e stabilizza il drone in base ai dati IMU e ai comandi di controllo PID.
- 2) **LEDs:** Viene acceso il LED verde per segnalare che il sistema è attivo e che i motori sono in esecuzione. Il LED rosso e giallo sono spenti.

3.4.3. Funzioni di Controllo dei Motori

- stopMotors()

La funzione stopMotors() spegne i motori impostando il duty cycle minimo per ciascun motore.

```
● ● ●  
53 void stopMotors()  
54 {  
55     setPwmNoControl(MIN_CALIB_DUTY, MIN_CALIB_DUTY, MIN_CALIB_DUTY, MIN_CALIB_DUTY);  
56 }
```

- armMotors()

La funzione armMotors() è utilizzata per preparare i motori all'attivazione. Essa imposta i motori a un valore di duty cycle fisso definito da ARM_DUTY.

```
● ● ●  
47 void armMotors()  
48 {  
49     setPwmNoControl(ARM_DUTY, ARM_DUTY, ARM_DUTY, ARM_DUTY);  
50 }
```

Entrambe le funzioni richiamano setPwmNoControl() che regola i PWM dei motori. Essa accetta quattro parametri, ciascuno dei quali corrisponde al duty cycle di uno dei motori. I valori passati vengono convertiti in valori di PWM proporzionali al valore di ARR (Auto Reload Register) del timer TIM3.

```
● ● ●  
47 void setPwmNoControl(float pwm1, float pwm2, float pwm3, float  
48     pwm4)  
49     TIM3->CCR1 = (uint32_t)(TIM3->ARR * pwm1/100);  
50     TIM3->CCR2 = (uint32_t)(TIM3->ARR * pwm2/100);  
51     TIM3->CCR3 = (uint32_t)(TIM3->ARR * pwm3/100);  
52     TIM3->CCR4 = (uint32_t)(TIM3->ARR * pwm4/100);  
53 }
```

N.B. Queste funzioni sono definite nel file ‘map.c’

- `stabilizeMotors()`

La funzione `stabilizeMotors()` è una delle funzioni principali del sistema, utilizzata per stabilizzare i motori del drone in base ai dati provenienti dal sensore IMU e alle leggi di controllo PID. Il suo scopo è quello di regolare i motori per mantenere il drone stabile, correggendo l'orientamento del drone in base ai comandi ricevuti. Si basa principalmente su:

1) Lettura dei Dati IMU

All'inizio della funzione, viene chiamata la funzione `readImu()`, che acquisisce i valori di orientamento del drone (roll, pitch, yaw) dai sensori IMU. La funzione `readImu()` restituisce i valori di orientamento che vengono utilizzati per il calcolo del controllo PID.

```

102 void readImu()
103 {
104     bno055_vector_t euler_vector = bno055_getVectorEuler();
105
106     roll = euler_vector.y;
107     if (euler_vector.z < 0)
108     {
109         pitch = -euler_vector.z - 180; //Se si raggiunge il SetPoint di -180° sul roll, questa
variabile vale 0
110     }
111     else{
112         pitch = -euler_vector.z + 180;
113     }
114     yaw = euler_vector.x;
115 }
```

2) Controllo dello Yaw

La funzione `stabilizeMotors()` verifica se la levetta di controllo dell'imbardata (`rif_yaw`) è cambiata rispetto al valore precedente (`prev_rif_yaw`). Se il valore è cambiato, viene calcolato un nuovo **yaw_setpoint** (angolo di yaw desiderato) aggiungendo un offset (`yaw_angolo`) all'attuale angolo di yaw (`yaw`). Il valore di **yaw_setpoint** viene normalizzato nell'intervallo [0, 360] per evitare valori fuori dal range. Se il valore di **yaw_setpoint** è stato raggiunto (la differenza tra il valore di **yaw_setpoint** e il valore corrente di **yaw** è inferiore a 1 grado), il controllo PID viene disattivato (modificando la variabile **flag_control**).

```

119 void stabilizeMotors()
120 {
121     float virtualInputs[4];
122     readImu();
123
124     //prev_rif_yaw non parte azzerato ad ogni ciclo, perché è dichiarato come static,
125     //quindi mantiene il suo valore tra le chiamate alla funzione
126     static int prev_rif_yaw = 0; // Memorizza il valore precedente della levetta
127     static int flag_control = 0; // Indica se il PID deve continuare a controllare
128
129     // Controllo cambio stato della levetta dell'imbarcatura
130     if (rif_yaw != prev_rif_yaw)
131     {
132         prev_rif_yaw = rif_yaw; // Aggiorna lo stato precedente
133         yaw_setpoint = yaw + yaw_angolo;
134
135         // Assicura che yaw_setpoint sia nel range [0, 360]
136         if (yaw_setpoint >= 360)
137             yaw_setpoint -= 360;
138         else if (yaw_setpoint < 0)
139             yaw_setpoint += 360;
140
141         flag_control = 1; // Attiva il controllo PID
142     }
143
144     // Se il PID ha raggiunto lo yaw_setpoint, ferma il controllo
145     //fabs() restituisce il valore assoluto di un numero in virgola mobile
146     if ((flag_control && fabs(yaw_setpoint - yaw) < 1.0)) // Tolleranza di 1 grado
147     {
148         flag_control = 0;
149     }

```

3) Controllo PID per Roll, Pitch e Yaw

Una volta calcolati i setpoint, la funzione calcola i Virtual Inputs per ogni asse. Vengono calcolati i controlli per i motori sulla base di Roll, Pitch, e Yaw:

- **Roll e Pitch:** vengono utilizzati i controlli PID per mantenere i valori di **roll** e **pitch** attorno al valore di equilibrio (0°).
- **Yaw:** Se il controllo per lo Yaw è attivo (ovvero se **flag_control** è impostato a 1), il PID per lo **yaw** viene calcolato e applicato solo se il drone si trova in una posizione di stabilità (all'interno di $\pm 12^\circ$ sia per **roll** che per **pitch**).

```

● ● ●

151 virtualInputs[0] = 9; // Peso del drone
152     virtualInputs[1] = PID_Controller(&RollPID, roll, 0);
153     virtualInputs[2] = PID_Controller(&PitchPID, pitch, 0);
154     //Attiva PitchYaw solo se il drone si trova in un intorno di +/-12° dalla posizione di
155     equilibrio
155     if (flag_control && (roll >= -12 && roll <= 12) && (pitch >= -12 && pitch <= 12))
156     {
157         virtualInputs[3] = PID_Controller(&YawPID, yaw, yaw_setpoint);
158     }
159     else
160     {
161         virtualInputs[3] = 0;
162     }

```

- 4) Calcolo della Velocità dei Motori e Conversione in PWM:** Una volta che gli input PID per Roll, Pitch e Yaw sono calcolati e memorizzati nell'array **virtualInputs[]**, questi vengono utilizzati dalla funzione **SpeedCompute()** per calcolare la velocità desiderata per ciascun motore. Le velocità ottenute vengono successivamente convertite in valori di duty cycle utilizzando la funzione **calcoloDuty()** per ciascun motore. I valori di duty cycle così calcolati vengono poi passati alla funzione **setPwm()**, che regola la velocità dei motori utilizzando il segnale PWM (Pulse Width Modulation).

```

● ● ●

164 float* Speeds;
165     Speeds = SpeedCompute(virtualInputs);
166
167 float avgMotor1 = calcoloDuty(*(Speeds+0));
168 float avgMotor2 = calcoloDuty(*(Speeds+1));
169 float avgMotor3 = calcoloDuty(*(Speeds+2));
170 float avgMotor4 = calcoloDuty(*(Speeds+3));
171
172 if(flag_print)
173 {
174     printf("avgMotor1: %.2f\r\n"
175             "avgMotor2: %.2f\r\n"
176             "avgMotor3: %.2f\r\n"
177             "avgMotor4: %.2f\r\n"
178             "Speed1: %.2f\r\n"
179             "Speed2: %.2f\r\n"
180             "Speed3: %.2f\r\n"
181             "Speed4: %.2f\r\n"
182             "roll: %f\r\n"
183             "pitch: %f\r\n"
184             "yaw: %f\r\n"
185             "rif_yaw: %d\r\n"
186             "yaw_setpoint: %.2f\r\n"
187             "flag_control: %d\r\n"
188             "Virtual Inputs: %f, %f, %f, %f\r\n",
189             avgMotor1, avgMotor2, avgMotor3, avgMotor4, Speeds[0], Speeds[1], Speeds[2], Speeds[3],
190             roll, pitch, yaw, rif_yaw, yaw_setpoint, flag_control,
191             virtualInputs[0], virtualInputs[1], virtualInputs[2], virtualInputs[3]);
192     flag_print=0;
193 }
194
195 setPwm(avgMotor1, avgMotor2, avgMotor3, avgMotor4);

```

3.4.4. Stima del coefficiente di drag

In questa sezione verrà illustrata la determinazione sperimentale del duty minimo per la rotazione in yaw e la stima del coefficiente d di resistenza aerodinamica.

Durante la fase di esecuzione (stato 2 del radiocomando), è stata attivata, solo in fase preliminare, una routine (attualmente commentata nel codice) finalizzata a determinare il *duty cycle minimo* necessario affinché il drone iniziasse a ruotare attorno all'asse yaw. Poiché il drone era vincolato a un supporto rigido, per avviare la rotazione era necessario superare l'attrito statico e la resistenza aerodinamica (drag).

La funzione `findDutyYaw()` incrementava progressivamente il duty cycle di due motori opposti, generando una coppia utile alla rotazione attorno all'asse verticale (yaw). Dopo ogni incremento, veniva letta l'IMU per verificare se l'angolo di yaw avesse subito una variazione superiore a una soglia predefinita (1.5°). Il primo valore di duty per cui si osservava una variazione sufficiente veniva registrato come *duty minimo* necessario a vincere l'insieme delle forze resistive.

A partire da questo duty minimo, è stata stimata la **velocità angolare del motore** (in RPM) utilizzando l'inversa della funzione che lega il duty alla velocità:

$$\text{speed} = \frac{(\text{MAX}_{\text{SPEED}} - \text{MIN}_{\text{SPEED}}) \cdot \text{duty} - (\text{MIN}_{\text{DUTY}} \cdot \text{MAX}_{\text{SPEED}} - \text{MAX}_{\text{DUTY}} \cdot \text{MIN}_{\text{SPEED}})}{\text{MAX}_{\text{DUTY}} - \text{MIN}_{\text{DUTY}}}$$

Successivamente, si è assunto che la **velocità dell'aria generata dalle eliche** fosse proporzionale alla velocità angolare, e si è utilizzata la relazione fisica diretta:

$$v = w \cdot r$$

La **velocità lineare dell'aria** così ottenuta è stata quindi utilizzata nell'equazione classica della resistenza aerodinamica, come riportato nella sezione *1.4.3 – Calcolo dei coefficienti aerodinamici*.

```

199 float findDutyYaw()
200 {
201     static float Duty_min_yaw = 0;
202     //Cambia solo una volta, quando il quadricottero inizia a spostarsi
203     float duty_1_3 = 5.4;
204     // Partiamo dal minimo valore armabile
205     float soglia = 1.5;
206     // Soglia di variazione dello yaw in gradi
207     double yaw_prev;
208
209     readImu();
210     yaw_prev = yaw;
211
212     while (duty_1_3 <= 6.8)
213     {
214         // Configuriamo il duty per generare una rotazione in yaw orario
215
216         setPwm(duty_1_3, 5.4, duty_1_3, 5.4);
217         HAL_Delay(500); // Aspettiamo mezzo secondo
218         readImu(); // Leggiamo il nuovo valore di yaw per vedere se il drone sta ruotando
219
220         // Se lo yaw cambia oltre la soglia, interrompiamo il ciclo
221         if (fabs(yaw - yaw_prev) > soglia)// Interrompe il test quando rileva una variazione di yaw
222             superiore a 1 grado
223         {
224             Duty_min_yaw = duty_1_3;
225             printf("Duty minimo per yaw: %f%\n", Duty_min_yaw);
226             break;
227         }
228         duty_1_3 += 0.2; // Incrementiamo il duty con step piccoli per precisione
229     }
230
231     // Rimettiamo i motori a 5.3%
232     setPwm(5.3, 5.3, 5.3, 5.3);
233
234     return Duty_min_yaw;
235 }
```

Nel ciclo while(1) è inoltre presente una variabile flag **calibrateFlag** che permette la calibrazione degli ESC quando viene attivata. La calibrazione viene effettuata tramite la funzione **ESC_Calibrate()**.

```

29 void ESC_Calibrate()
30 {
31     HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0, GPIO_PIN_SET); //Led verde
32     setPwmNoControl(MAX_CALIB_DUTY, MAX_CALIB_DUTY, MAX_CALIB_DUTY, MAX_CALIB_DUTY); //10
33     HAL_Delay(3000); //ATTACCARE BATTERIA PRIMA CHE SI SPENGA IL LED VERDE
34     HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0, GPIO_PIN_RESET);
35     setPwmNoControl(MIN_CALIB_DUTY, MIN_CALIB_DUTY, MIN_CALIB_DUTY, MIN_CALIB_DUTY); //5
36 }
```

4. Test e Simulazioni

Durante la realizzazione del progetto abbiamo eseguito vari test e simulazioni, inizialmente per studiare e comprendere poco per volta l'intero sistema del drone, sia a livello meccanico (struttura fisica e componenti) che di funzionamento (codice per l'acquisizione dati e per il controllo dei motori), e in seguito per migliorare la stabilità sul piano xy e implementare la rotazione intorno all'asse z.

In particolare, durante lo studio e lo sviluppo del codice per la stabilità e la rotazione, abbiamo dedicato gran parte del tempo alla taratura dei tre PID di Roll, Pitch e Yaw, controllando il loro andamento attraverso dei grafici generati su Matlab.

Le prime simulazioni sulla taratura dei soli PID di Roll e Pitch sono state fatte senza fissare ai lati del tavolo le quattro corde legate ai sostegni dei motori, in modo da non avere poi vincoli per l'imbardata. Successivamente abbiamo però ricominciato i test nelle stesse condizioni del gruppo che ci aveva lavorato precedentemente, ossia fissando le corde ai quattro lati del tavolo, per limitare i movimenti bruschi del drone e quindi le forti oscillazioni rilevate dal sensore IMU. In questo modo, senza la necessità di tarare nuovamente i PID di Roll e Pitch, abbiamo direttamente utilizzato i coefficienti impostati dall'altro gruppo e siamo passati alla taratura del PID di Yaw.

4.1. Codice MATLAB

Abbiamo utilizzato MATLAB per acquisire i dati dal drone e tracciare vari grafici. Dai printf() della funzione stabilizeMotors(); vengono stampati nella seriale diversi dati, in particolare:

- (avgMotor1-4) i quattro duty da impostare ai motori;
- (roll, pitch, yaw) gli angoli di Eulero letti dalla IMU;
- (VirtualInputs2-4) gli output dei tre PID.

Acquisiti tali valori, si vanno poi a generare quattro grafici per valutarne l'andamento.

Codice in acquisizione_dati.m

```
% Pulizia della workspace e chiusura di eventuali connessioni seriali aperte
clear; clc; close all;

% Configurazione della porta seriale
port = "COM4";
baudRate = 115200;
portaSeriale = serialport(port, baudRate);
configureTerminator(portaSeriale, "CR/LF"); % STM32 invia dati con "\r\n"
flush(portaSeriale); % Pulisce il buffer

disp("Connessione alla porta seriale stabilita.");

% Creazione e apertura file CSV per la scrittura
fileName = "dati_drone.csv";
fid = fopen(fileName, 'w');
fprintf(fid, "Tempo,Roll,Pitch,Yaw,RifYaw,YawSetpoint," + ...
    "VirtualInput1,VirtualInput2,VirtualInput3,VirtualInput4," + ...
    "AvgMotor1,AvgMotor2,AvgMotor3,AvgMotor4\n");

% Inizializzazione variabili
roll = 0; pitch = 0; yaw = 0; rif_yaw = 0; yaw_setpoint = 0;
avgMotor1 = 0; avgMotor2 = 0; avgMotor3 = 0; avgMotor4 = 0;
VirtualInput1 = 0; VirtualInput2 = 0; VirtualInput3 = 0; VirtualInput4 = 0;

% Avvio del timer
startTime = tic;
numSamples = 20000; % 200 secondi con frequenza di 10 Hz
sampleCount = 0;

disp("Inizio acquisizione dati...");
while sampleCount < numSamples
    if portaSeriale.NumBytesAvailable > 0
        data_line = readline(portaSeriale);
        disp(data_line); % Mostra i dati ricevuti per debugging

        % Estrai i valori dalle righe ricevute
        if contains(data_line, "avgMotor1:")
            avgMotor1 = str2double(extractAfter(data_line, "avgMotor1: "));
        elseif contains(data_line, "avgMotor2:")
            avgMotor2 = str2double(extractAfter(data_line, "avgMotor2: "));
        elseif contains(data_line, "avgMotor3:")
            avgMotor3 = str2double(extractAfter(data_line, "avgMotor3: "));
        elseif contains(data_line, "avgMotor4:")
            avgMotor4 = str2double(extractAfter(data_line, "avgMotor4: "));
        elseif contains(data_line, "roll:")
            roll = str2double(extractAfter(data_line, "roll: "));
        elseif contains(data_line, "pitch:")
            pitch = str2double(extractAfter(data_line, "pitch: "));
        elseif contains(data_line, "yaw:")
            yaw = str2double(extractAfter(data_line, "yaw: "));
        elseif contains(data_line, "rif_yaw:")
            rif_yaw = str2double(extractAfter(data_line, "rif_yaw: "));
        elseif contains(data_line, "yaw_setpoint:")
            yaw_setpoint = str2double(extractAfter(data_line, "yaw_setpoint: "));
        elseif contains(data_line, "Virtual Inputs:")
            values = sscanf(extractAfter(data_line, "Virtual Inputs: "), "%f, %f, %f, %f");
            if numel(values) == 4
                VirtualInput1 = values(1);
                VirtualInput2 = values(2);
                VirtualInput3 = values(3);
                VirtualInput4 = values(4);
            end
        end
    end
end
```

```

% Tempo trascorso
elapsedTime = toc(startTime);

% Scrivi i dati nel file CSV
fprintf(fid, "%.3f,%.3f,%.3f,%.3f,%d,%.3f,%.3f,%.3f,%.3f,%.3f,%.3f,%.3f,%.3f,%.3f,%.3f\n",
        elapsedTime, roll, pitch, yaw, rif_yaw, yaw_setpoint, ...
        VirtualInput1, VirtualInput2, VirtualInput3, VirtualInput4, ...
        avgMotor1, avgMotor2, avgMotor3, avgMotor4);

% Incrementa il conteggio dei campioni
sampleCount = sampleCount + 1;
end
end

% Chiudi il file e la connessione seriale
fclose(fid);
clear portaSeriale;
disp("Acquisizione completata. Dati salvati in dati_drone.csv");

```

Codice in grafico_dati.m

```

% Pulizia della workspace
clear; clc; close all;

% Caricamento dei dati dal CSV
fileName = "dati_drone.csv";
dati = readtable(fileName);

% Estrai i dati dalle colonne
tempo = dati.Tempo;
roll = dati.Roll;
pitch = dati.Pitch;
yaw = dati.Yaw;
rif_yaw = dati.RifYaw;
yaw_setpoint = dati.YawSetpoint;
VirtualInput1 = dati.VirtualInput1;
VirtualInput2 = dati.VirtualInput2;
VirtualInput3 = dati.VirtualInput3;
VirtualInput4 = dati.VirtualInput4;
avgMotor1 = dati.AvgMotor1;
avgMotor2 = dati.AvgMotor2;
avgMotor3 = dati.AvgMotor3;
avgMotor4 = dati.AvgMotor4;

% Creazione della figura con 4 subplot
figure;

% 1. Roll, Pitch e Yaw
subplot(4,1,1); hold on;
plot(tempo, roll, 'r', 'LineWidth', 1.5);
plot(tempo, pitch, 'g', 'LineWidth', 1.5);
plot(tempo, yaw, 'b', 'LineWidth', 1.5, 'Color', [0 0 1 0.2]); % Blu con 60% di opacità
xlabel("Tempo (s)");
ylabel("Angoli (°)");
title("PITCH, ROLL e YAW");
legend("Roll", "Pitch", "Yaw");
grid on;

```

```

% 2. Virtual Inputs
subplot(4,1,2); hold on;
%plot(tempo, VirtualInput1, 'k', 'LineWidth', 1.5);
plot(tempo, VirtualInput2, 'b', 'LineWidth', 1.5);
plot(tempo, VirtualInput3, 'g', 'LineWidth', 1.5);
plot(tempo, VirtualInput4, 'r', 'LineWidth', 1.5);
xlabel("Tempo (s)"); ylabel("Virtual Inputs");
title("Virtual Inputs");
legend("VirtualInput2", "VirtualInput3", "VirtualInput4");
grid on;

% 3. Motori 1 e 3
subplot(4,1,3); hold on;
plot(tempo, avgMotor1, 'b', 'LineWidth', 1.5);
plot(tempo, avgMotor3, 'r', 'LineWidth', 1.5);
xlabel("Tempo (s)"); ylabel("AvgMotor");
title("AVGMotor 1-3");
legend("AvgMotor1", "AvgMotor3");
grid on;

% 4. Motori 2 e 4
subplot(4,1,4); hold on;
plot(tempo, avgMotor2, 'b', 'LineWidth', 1.5);
plot(tempo, avgMotor4, 'r', 'LineWidth', 1.5);
xlabel("Tempo (s)"); ylabel("AvgMotor");
title("AVGMotor 2-4");
legend("AvgMotor2", "AvgMotor4");
grid on;

```

4.2. Processo di taratura

Come spiegato in precedenza, il regolatore PID è un algoritmo di controllo a retroazione progettato per mantenere una variabile di processo il più vicina possibile a un valore desiderato.

La funzione `PID_Controller()` utilizzata segue questo principio, operando come un sistema di controllo a feedback che calcola continuamente l'**errore**, ovvero la differenza tra il valore desiderato (`setPoint`) e la variabile di processo attuale (`input`). In base a questo errore, il PID va a calcolare il valore della variabile di controllo (`output`, assegnata a `VirtualInputs[1-3]`) come somma di tre componenti (`Pterm + newIterm + Dterm`), regolando di conseguenza la velocità dei motori per stabilizzare il drone e guidarlo verso il comportamento desiderato.

Per ottenere un controllo efficace, è necessario trovare i coefficienti K_p , K_i e K_d , che determinano il peso di ciascuna componente nella risposta del sistema.

Ognuna delle tre componenti del PID ha un ruolo specifico nel controllo del sistema:

- **Termine Proporzionale:** genera un'azione correttiva proporzionale all'errore. Maggiore è il guadagno proporzionale K_p , maggiore è la reattività del sistema, ma un valore troppo alto può causare instabilità e oscillazioni.
- **Termine Integrale:** accumula l'errore nel tempo per eliminare l'offset residuo che il solo termine proporzionale lascerebbe. Tuttavia, un K_i troppo alto può provocare sovraelongazione (overshoot) e instabilità, a causa del fenomeno noto come windup integrale (accumulo di un errore eccessivo quando l'output è saturato, Ex: raggiunto MaxDuty dei motori).
- **Termine Derivativo:** calcola la velocità di variazione dell'errore e applica una correzione per smorzare le oscillazioni. Se K_d è troppo elevato, può amplificare il rumore e rendere il sistema instabile.

Nel nostro caso, la priorità era ottenere una risposta **reattiva** per contrastare rapidamente l'errore di inclinazione dovuto alla gravità, evitando però oscillazioni eccessive.

Metodo trial-and-error

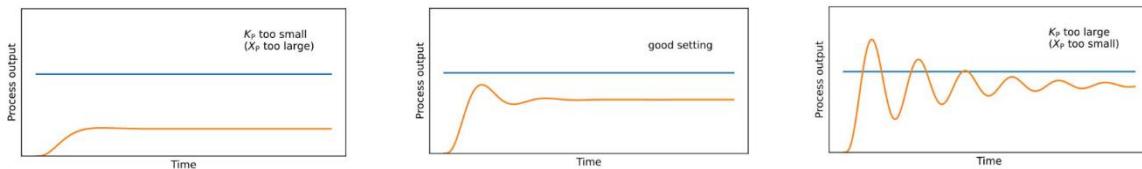
Per la taratura del PID abbiamo adottato un approccio empirico basato sul metodo trial-and-error. Il principio alla base di questa tecnica di regolazione per tentativi è che la taratura del PID può essere fatta regolando progressivamente l'aggressività delle singole azioni, partendo dalla componente proporzionale, per poi passare a quella integrale e infine a quella derivativa, fino a raggiungere il comportamento desiderato.

Il processo si svolge nei seguenti passi:

- 1) **Taratura della sola componente poporzionale K_p :** disattivare le azioni integrale e derivativa e regolare solo il guadagno proporzionale K_p :

- iniziare con un valore basso di K_p , scelto in base all'ordine di grandezza con cui una variazione della variabile manipolata (output del PID, che modifica la velocità dei motori) influisce sulla variabile controllata (angoli di inclinazione misurati dalla IMU). È necessario quindi fare una stima di quanto varia l'angolo di inclinazione del drone quando si modifica l'uscita del PID;
- aumentare progressivamente K_p fino a ottenere una risposta sufficientemente rapida. Con l'aumento del guadagno, l'errore di offset si riduce, ma possono comparire oscillazioni;
- individuare il limite oltre il quale il sistema diventa instabile: quando la risposta mostra oscillazioni persistenti o il sistema diverge, K_p è eccessivo.

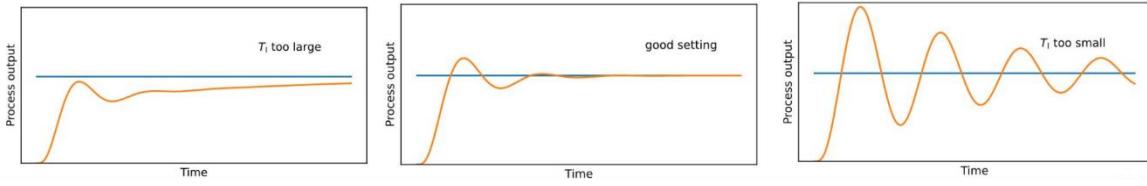
Un buon valore di taratura si ha quando è presente un evidente overshoot, ma che si smorza rapidamente.



2) **Aggiunta della componente integrale K_i :** dopo aver regolato K_p , si introduce K_i per eliminare l'errore di offset residuo:

- iniziare con un valore basso di K_i . Un buon punto di partenza può essere l'inverso della costante di tempo del sistema (stima del tempo (T_i) necessario affinché il processo si stabilizzi dopo un'azione di controllo);
- aumentare progressivamente K_i , osservando la velocità con cui l'errore si annulla. Come per l'azione proporzionale, un'azione integrale troppo aggressiva porta a oscillazioni indesiderate;
- continuare fino a raggiungere il punto in cui la risposta del sistema raggiunge il setpoint desiderato con la velocità desiderata.

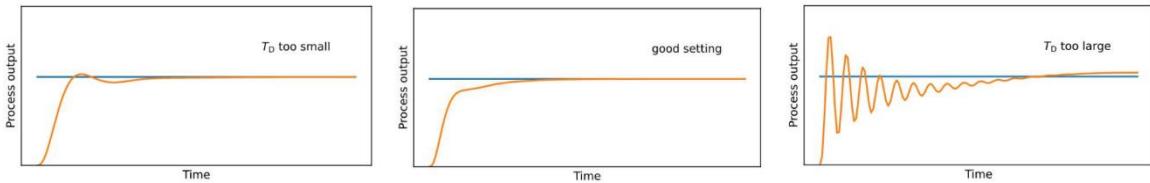
Per molte applicazioni, un regolatore PI ben tarato è sufficiente.



3) **Aggiunta della componente derivativa Kd:** ripetere la stessa procedura anche per l'azione derivativa, al fine di migliorare la risposta del sistema riducendo overshoot e oscillazioni:

- iniziare con un valore basso di Kd. Un buon punto di partenza può essere un decimo del tempo di integrazione (Ti) precedentemente selezionato per la componente integrale;
- effettuare scatti improvvisi del setpoint e osservare la risposta del sistema. Se ci sono ancora oscillazioni, aumentare progressivamente Kd;
- continuare a regolare Kd fino a trovare un compromesso tra stabilità e velocità di risposta.

Un vantaggio della componente derivativa è che consente di adottare valori più elevati di Kp senza compromettere la stabilità del sistema. Per questo motivo, a questo punto può essere utile ricalibrare leggermente Kp per ottenere un controllo più reattivo.



Riassumendo: inizialmente abbiamo disabilitato i termini Ki e Kd, regolando solo Kp per ottenere una risposta rapida. Una volta ottenuto un valore soddisfacente, abbiamo aggiunto Ki per ridurre eventuali errori di offset. Infine, abbiamo introdotto Kd per ridurre le oscillazioni residue.

4.3. Taratura PID di Roll e Pitch

Per garantire la stabilità del quadricottero sul piano xy, abbiamo inizialmente regolato solo i PID di Roll e Pitch. Quindi impostati i parametri del PID di Yaw a zero, abbiamo cominciato dai termini proporzionali.

Considerando che in seguito avremo dovuto implementare anche il movimento di imbardata, abbiamo inizialmente optato per tarare i PID tenendo i quattro cavi dei bracci del drone slegati dai lati del tavolo.

Quindi siamo partiti dagli stessi valori di Kp ($K_{pr} = K_{pp} = 0.016$) e del peso (virtualInputs[0] = 11) lasciati dal gruppo precedente.

Prima fase: regolazione Kp

Abbiamo settato $K_{ir} = K_{ip} = 0$ e $K_{dr} = K_{dp} = 0$, lavorando solo su Kp.

- **Test 1:** con $K_{pr} = K_{pp} = 0.016$ il drone risultava troppo poco reattivo, e non riuscendo a contrastare la gravità rimaneva infine fermo in posizione inclinata.



- **Test 2:** aumentando Kp a 0.019, la risposta è diventata eccessivamente reattiva, causando oscillazioni troppo marcate.



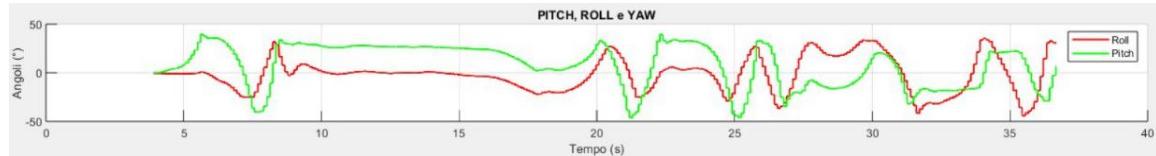
- **Test 3:** un valore intermedio di Kp = 0.018 ha permesso di ottenere una risposta più equilibrata, ma con oscillazioni iniziali e ampio offset, suggerendo la necessità di introdurre Ki.



Seconda fase: regolazione Ki

Abbiamo quindi introdotto il termine integrale per eliminare l'offset residuo:

- **Test 5:** valutando i valori impostati dal gruppo precedente, siamo partiti con $K_{ir} = K_{ip} = 0.0001$. Inizialmente il sistema sembrava stabile, ma le oscillazioni sono riprese dopo qualche secondo.



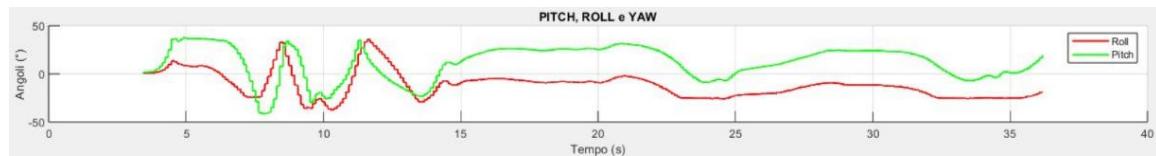
- **Test 6:** abbiamo provato a ridurre K_i di un ordine di grandezza $K_{ir} = K_{ip} = 0.00001$, il drone ha mostrato meno oscillazioni rispetto al test precedente.



Terza fase: regolazione Kd

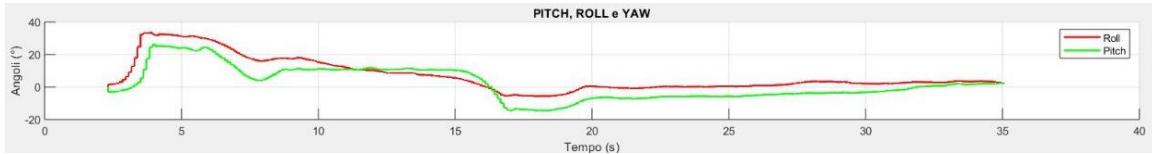
Infine, abbiamo introdotto il termine derivativo per smorzare ulteriormente le oscillazioni:

- **Test 7:** introducendo $K_{dr} = K_{dp} = 0.05$, le oscillazioni iniziali non si sono attenuate come previsto.



- **Test 16:** dopo qualche altro test non troppo soddisfacente abbiamo deciso di diminuire il peso del drone a 9 (`virtualInputs[0] = 9`) per limitare le brusche variazioni di velocità dei motori. Inoltre, come testato dal gruppo precedente, abbiamo rimesso i vincoli dei cavi collegati al tavolo per evitare grandi sbilanciamenti dei bracci del drone. Abbiamo poi ridotto $K_p = 0.016$ per evitare un eccesso di reattività, ripristinato K_i al vecchio ordine di grandezza (0.0001) per

correggere meglio l'offset e infine aumentato $K_d = 0.06$ per stabilizzare il sistema, ottenendo un buon compromesso.



I valori finali impostati per i PID di Roll e Pitch sono stati:

- $K_{pr} = 0.016$, $k_{pp} = 0.016$
- $K_{ir} = 0.0001$, $K_{ip} = 0.0002$
- $K_{dr} = 0.06$, $k_{dp} = 0.06$

Con questi valori, il sistema ha mostrato una buona stabilità e una riduzione significativa delle oscillazioni residue, come confermato dall'ultimo grafico in MATLAB.

4.4. Taratura PID di Yaw

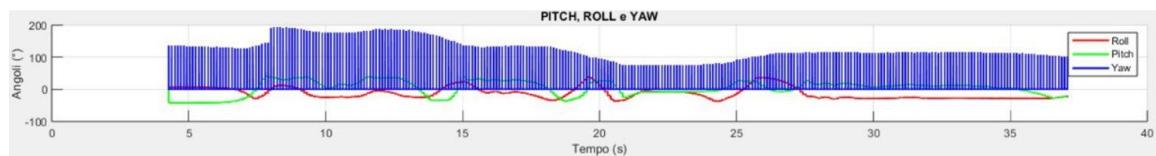
Anche qui, la taratura è stata fatta in modo empirico. Abbiamo inizialmente impostato solo il termine proporzionale per ottenere una risposta sufficientemente reattiva, poi abbiamo testato il termine integrale per correggere eventuali errori accumulati. Infine, abbiamo valutato il termine derivativo, che si è rivelato poco utile a causa della natura del controllo dell'imbardata.

Leva dello Yaw: inizialmente abbiamo tarato il PID di Yaw mantanendo yaw_setpoint = yaw, in modo tale da studiare il suo contributo alla stabilità del drone in assenza di rotazione intorno all'asse z. In seguito abbiamo ripetuto i test comprendendo anche il movmento di imbardata, ossia regolando il SetPoint attraverso lo stato della leva sinistra del radiocomando (rif_yaw), la quale comanda una rotazione di 20° (yaw_angolo) in senso orario quando viene posizionata verso l'alto, di 20° in senso antiorario quando si trova verso il basso, e nessuna rotazione quando si trova al centro.

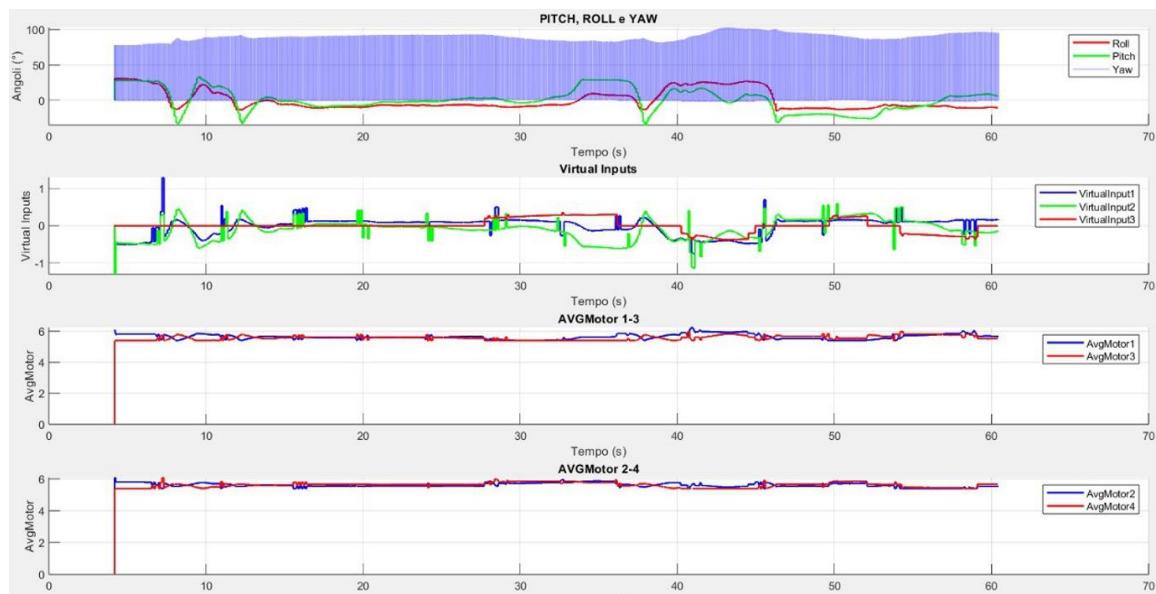
Prima fase: regolazione Kp

Abbiamo settato $K_{iy} = K_{dy} = 0$, lavorando solo su K_p .

- **Test 18:** mantenendo attivi i PID di Roll e Pitch e attivando il PID di Yaw con $yaw_setpoint = yaw$, siamo andati a regolare K_p , ponendo inizialmente $K_{py} = 0.01$. Questo terzo PID ha aiutato a mantenere la stabilità del drone sul piano xy, evitando rotazioni intorno all'asse z. Da notare che i valori di yaw hanno escursioni molto ampie poiché l'angolo di yaw varia da 0° a 360° , quindi se il drone oscilla intorno alla posizione di 0° può registrare valori di yaw che vanno da circa 10° a 350° ($\pm 10^\circ$ dalla posizione di 0°).



- **Test 21:** attivando anche la leva di Yaw del radiocomando per variare $yaw_setpoint$ ($\pm 20^\circ$) e indurre il movimento di imbardata, abbiamo ottenuto buoni risultati, mostrando una risposta reattiva e priva di oscillazioni indesiderate, con una buona stabilizzazione durante la rotazione.



I parametri finali del PID di Yaw sono stati:

- $K_{py} = 0.04$
- $K_{iy} = 0.00$
- $K_{dy} = 0.00$

Considerando che le eliche dei motori **1 e 3** ruotano in **senso orario** e quelle dei motori **2 e 4** in **senso antiorario**, quando la leva di yaw del radiocomando viene spinta in alto (`rif_yaw = 1`), il drone dovrebbe ruotare in senso orario e, viceversa, in senso antiorario quando la leva è in basso (`rif_yaw = -1`).

Guardando l'ultimo grafico (Test 21) si può notare una buona risposta del sistema in seguito ai comandi di imbardata attivati dalla leva di yaw. L'andamento delle velocità dei motori è indicato dai grafici di AvgMotors, mentre lo stato della leva di Yaw può essere ricavato dall'andamento del VirtualInputs[3] del PID di Yaw (che in questo test varia da -0.5 a 0.5):

- A circa 28s, viene applicato un comando di imbardata positiva (`rif_yaw = 1`), che causa un aumento della velocità dei motori 2 e 4 e una riduzione della velocità dei motori 1 e 3. Il risultato è una rotazione oraria del drone (da notare che lo yaw varia da circa 90° a 70°).
- A circa 40s, viene applicato un comando di imbardata negativa (`rif_yaw = -1`), di conseguenza i motori 1 e 3 aumentano la velocità mentre i motori 2 e 4 la riducono, causando una rotazione antioraria del drone (yaw varia da circa 80° a 100°).

Tra i movimenti di imbardata ci sono intervalli in cui il drone mantiene la stabilità senza ruotare (`rif_yaw = 0`).

Conclusione e Osservazioni finali

Nel complesso il progetto ha portato allo sviluppo di un sistema funzionante per il controllo e la stabilizzazione del quadricottero. Il drone risponde correttamente ai comandi base di accensione, armamento e spegnimento, e gestisce in modo efficace anche i comandi di imbardata impartiti tramite la leva di yaw. Una volta attivato, il sistema è in grado di stabilizzare il drone sul piano xy in tempi brevi e consente una rotazione ben visibile intorno all'asse z. Durante lo sviluppo, tuttavia, sono emerse alcune criticità legate principalmente alla struttura fisica del drone, all'acquisizione dei dati e alla precisione della modellazione:

- **Coefficienti b e d non realistici:** i valori inizialmente scelti per la spinta S e per la velocità angolare ω dell'elica nel calcolo del coefficiente di spinta b non rispecchiavano le reali condizioni operative del drone. Questo ha portato a un valore di b più basso rispetto a quello realistico e, di conseguenza, il calcolo del coefficiente di resistenza aerodinamica d tramite la funzione `findDutyYaw()` è stato abbandonato in favore di un approccio empirico che tenesse conto del valore assegnato a b .
- **Oscillazioni eccessive rilevate dalla IMU:** nonostante l'acquisizione dati sia in modalità “fast mode”, la IMU rileva oscillazioni dovute a vibrazioni meccaniche e alla scarsa rigidità strutturale del drone. Il supporto centrale su cui è montato tende a vibrare durante il funzionamento, e si è osservato che il posizionamento di oggetti pesanti sopra la base migliora lievemente la stabilità.
- **Composizione fisica non ottimale:** alcuni componenti non sono fissati saldamente, contribuendo ulteriormente alle oscillazioni. In particolare, la batteria, fissata con un lacchetto a strappo, tende a scivolare durante l'esecuzione, perciò abbiamo cercato di migliorarne il fissaggio con l'aggiunta di nastro adesivo disposto in modo perpendicolare al lacchetto. L'attuale supporto in plastica, sede dei componenti e sensori principali, essendo troppo alto, agisce come una leva meccanica che amplifica le vibrazioni e i disturbi. Si consiglia di abbassare il supporto o utilizzare materiali smorzanti per ridurre queste sollecitazioni.

- **IMU non posizionata nel baricentro:** il sensore è montato in una posizione asimmetrica rispetto al centro del drone, con orientamento dell'asse z verso il basso. Questo ha influito sulla simmetria delle letture, rendendo necessario invertire i coefficienti di controllo dello yaw nella funzione SpeedCompute() rispetto a quanto previsto dal modello matematico.
- **Difficoltà nella taratura dei controllori PID:** non è stato possibile adottare un approccio metodologico per la taratura dei PID, in parte a causa delle problematiche strutturali e ai disturbi, perciò abbiamo optato per una taratura empirica.

Nonostante le limitazioni fisiche e le difficoltà incontrate in fase di taratura, il progetto ha raggiunto gli obiettivi principali. Il lavoro svolto costituisce una solida base per futuri miglioramenti e test in condizioni più realistiche. L'esperienza ha permesso di comprendere a fondo sia la complessità della modellazione teorica che le sfide pratiche dell'implementazione su un sistema hardware reale.

Bibliografia

- Turnigy Manual for Brushless Motor Speed Controller (<https://hbfpnl.wordpress.com/wp-content/uploads/2009/11/turnigy-plush-esc-user-manual.pdf>)
- IMU BNO055 datasheet (https://cdn-shop.adafruit.com/datasheets/BST_BNO055_DS000_12.pdf)
- Altri riferimenti alla teoria, in particolare quella riguardante PWM e PID, e caratteristiche dei componenti sono stati presi dalle slide delle lezioni e dalla relazione del progetto precedente.

Link al nostro repository GitHub con il codice e la documentazione raccolta:
<https://github.com/Slesiac/Drone-Automation-Lab>