DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE E
SCIENZE MATEMATICHE

Corso di laurea in
Ingegneria Informatica e dell'Informazione

Modellizzazione e controllo del drone quadrirotore Crazyflie 2.1

Relatore:
Prof. Gianni Bianchini

Candidato:
Edoardo Caproni

Anno Accademico 2022 – 2023

Indice

Introduzione

Scopo di questa tesi è lo studio e il controllo della piattaforma Crazyflie 2.1, un quadritotore prodotto da Bitcraze, caratterizzato da bassi costi, e codice open-source (sia il firmware, che l'applicazione software – cfclient –, che l'interfaccia API). Queste caratteristiche rendono la piattaforma particolarmente adatta a fini di ricerca e sperimentazione. Degno di nota è anche il supporto offerto nel forum ufficiale[i], e l'integrazione di critiche e progetti offerti dalla comunità, che hanno notevolmente contribuito alla qualità finale del prodotto.

Nel nostro caso ci interessa progettare una struttura di controllo adeguata alla stabilizzazione di volo e inseguimento di traiettoria, e implementarla off-board con Python, utilizzando i mezzi forniti dall'API per dialogare con il drone.

Capitolo 1 - Specifiche della piattaforma

Crazyflie 2.1

Crazyflie 2.1 (Figura 1) è un drone quadrirotore offerto da Bitcraze, azienda nata in Svezia, che si propone l'obiettivo di offrire una macchina volante con un basso numero di componenti meccaniche, e sia di dimensioni contenute, facile da assemblare, consenta il volo indoors e sia gestita da software open-source. Il prodotto raggiunge tutti questi obiettivi, che rappresentano però importanti trade-off in altre aree: ad esempio piccolezza ed economicità delle componenti implicano una struttura flessibile e quindi fortemente affetta da vibrazioni.

Figura 1 - Crazyflie 2.1 a fine assemblaggio

Il drone utilizza come frame la stessa PCB (printed circuit board) che contiene i microcontrollori e chip sensoriali; quattro scocche di plastica offrono alloggi ad altrettanti motori, e si agganciano al frame a incastro.

I quattro motori sono coreless, con un diametro di 7 mm. Lavorano in corrente continua, il che li rende controllabili con PWM (pulse-width modulation, o modulazione di larghezza d'impulso), che consente di simulare diversi livelli di tensione e quindi di velocità di rotazione. Le eliche sono di plastica, e appaiate in coppie di rotazione una oraria e una antioraria su diagonali opposte; questo per indurre un momento di imbardata nullo per somma algebrica, supposti identici i momenti angolari di ogni motore. Precise istruzioni di assemblaggio possono essere trovate sul sito della Bitcraze.ii

La struttura risultante non è perfettamente rigida, ma offre una notevole resistenza ad urti e cadute: Bitcraze vanta test di caduta da 30 metri che riportano danni ai soli motori, eliche, o scocche di plastica, ma non alla PCB, essendo fatta di FR-4 (un materiale in fibra di vetro piuttosto resistente).[iii]

Le dimensioni sono molto ridotte, appena 92 mm x 92 mm x 29 mm una volta assemblato, con un peso di 27 g e un payload raccomandato di 15 g.[iv] Questo implica una certa attenzione nella scelta di estensioni hardware del drone, che possono essere acquistate dall'azienda sotto forma di deck di espansione, o progettate dall'utente.

La batteria stock è una LiPo (litio-polimero) da 250 mAh; consente un tempo di volo non superiore ai 7 minuti, a fronte di 40 minuti di ricarica, limitando fortemente le opzioni di volo.

Il drone è fornito di una radio sulla banda ISM (Industrial, Scientific & Medical) – canale di 2.4GHz – per comunicare con un computer dotato di Crazyradio PA, un dongle USB basato su nRF24LU1+ (un ricetrasmettitore della Nordic Semiconductors).[v] Grazie ad un'amplificazione di 20 dBm (decibell milliwatt) vanta un range superiore al kilometro supposte condizioni ideali, quindi decisamente appropriato per utilizzo domestico. Da menzionare anche la possibilità di interfacciarsi attraverso Bluetooth, opzione usata dall'applicazione di comando da dispositivi mobili.

Un altro approccio di comunicazione con il drone è attraverso la porta micro-USB, che è usata per la ricarica della batteria e il flashing del firmware.

I sensori installati sono un giroscopio/accelerometro a 3 assi (BMI088) e un sensore di pressione di alta precisione (BMP388), che apre la possibilità di utilizzo ad alte quote. Giroscopi e accelerometri hanno intrinsecamente difficoltà nel fornire letture precise di angolo e posizione rispettivamente, poiché partono da una rilevazione in accelerazione, su cui viene applicata una doppia integrazione. Questo genera errori, di cui particolarmente dannosi sono quelli di deriva.

Per correggere errori di deriva, è stato aggiunto un blocco di espansione, il FlowDeck V2. Questo include due sensori: il VL53L1x, una telecamera a tempo di volo (Time of Flight) che misura la distanza dalla superficie sottostante, e il PMW3901, un sensore di flusso ottico (optical flow, da cui il nome del deck), che compara fotorilevazioni del terreno per computare gli spostamenti orizzontali relativi ad esso. Il sensore di flusso necessita di almeno 80 mm di quota per fornire letture precise, e ha un cono di visione di 42° che bisogna aver cura di non ostruire. È altresì importante controllare che la superficie su cui far volare il drone sia piatta per un corretto utilizzo del lettore di quota, e che non presenti texture che possano confondere il sensore di flusso (ad esempio superfici perfettamente monocrome in quanto non offrono punti di riferimento, o molto "rumorose" come un ghiaino).

Le letture dei sensori sono mediate da un filtro di Kalman esteso, che può essere configurato liberamente nel firmware del Crazyflie.vi Da notare che le misure restano comunque affette da errori, sopratutto di deriva nello yaw/imbardata; una correzione più puntuale si può ottenere attraverso l'uso di altri deck di espansione (come Loco Positioning o Lighthouse Positioning), oppure con l'integrazione di un sistema di motion capture.

Capitolo 2 - Descrizione del modello dinamico

È importante avere una descrizione della dinamica del moto del drone per capire la complessità del sistema da controllare e le sue non linearità, per individuare le variabili di stato e scegliere conseguentemente quali letture sensoriali debbano essere raccolte e registrate dal drone.

2.1 - SISTEMA DI RIFERIMENTO

Il Crazyflie utilizza un sistema di riferimento nello standard ENU, ovvero east-north-up, sia per il sistema di coordinate globale (o terra o world), sia per quello locale (o body-fixed). Gli assi x-y-z sono infatti associati a quest'ordine di direzioni.vii L'orientamento del drone è rappresentato in angoli di Eulero nello schema 1-2-3 (o x-y-z); ovvero le rotazioni attorno agli assi avvengono nell'ordine presentato. La particolarità però è che gli angoli di roll e yaw (rollio e imbardata) sono rotazioni positive in senso orario osservando l'asse dall'origine, mentre l'angolo di pitch (beccheggio) è positivo in senso antiorario (Figura 2). Questa eccezione è importante da tenere a mente nell'interpretazione delle letture di angolo dal drone, e nel calcolo di eventuali matrici di rotazione. L'origine di questa scelta si trova in codice legacy, e non è mai stata modificata per non perdere compatibilità.

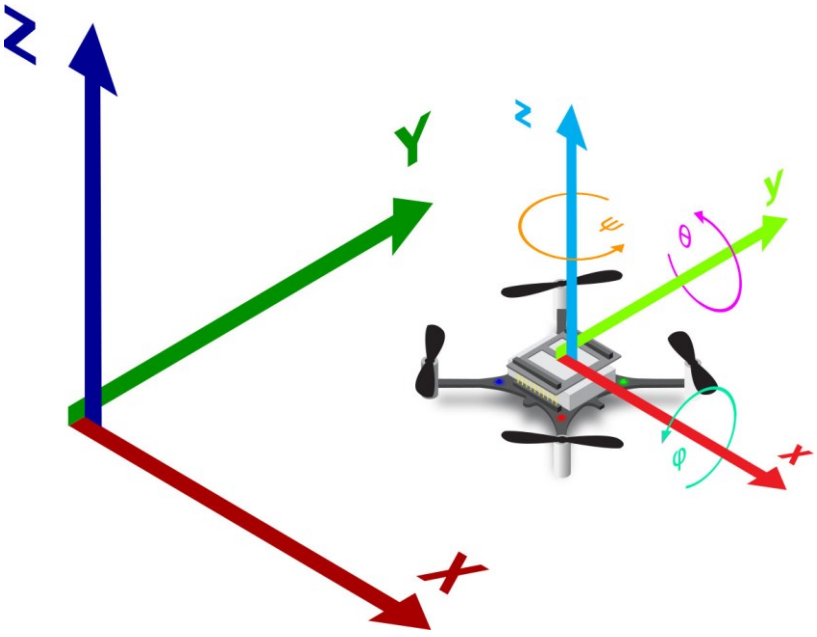

Figura 2 - Definizione sistema di riferimento

2.2 - VARIABILI DI CONTROLLO

Per poter raggiungere una qualunque combinazione di posizione ed orientamento è necessario capire come governare il drone in modo da ottenere reazioni elementari sommabili linearmente. Si nota subito che il sistema è sottoazionato, in quanto dotato di quattro attuatori (i motori) ma sei gradi di libertà: evidentemente rollio e beccheggio saranno accoppiati con movimenti lungo gli assi y e x rispettivamente, rendendo impossibile inseguire una traiettoria arbitraria nello spazio degli stati. Banalmente, non è possibile inclinarsi in una direzione e spostarsi in quella opposta.

Gli ingressi del sistema saranno quindi associati ai rimanenti quattro gradi di libertà, e quindi saranno quattro comandi linearmente sovrapponibili. Ovvero:

U1 : thrust (spinta), cioè la forza lungo l'asse z (nel sistema di riferimento del drone), che viene attuata ripartendola equamente tra i quattro motori;

U2 : momento di roll (rollio), attuato aggiungendo potenza ai motori sul lato sinistro e riducendola a quelli sul destro se positivo (o viceversa);

U3 : momento di pitch (beccheggio), attuato aggiungendo potenza ai motori posteriori e riducendola a quelli frontali se positivo (o viceversa);

U4 : momento di yaw (imbardata), che viene attuato intensificando la coppia di eliche oraria (se negativo) o antioraria (se positivo), e indebolendo l'altra.

Un altro standard prevede di assegnare rollio e beccheggio a solo due motori opposti rispetto alla diagonale, invece che a due coppie di motori opposte rispetto al lato: questo produce un sistema di coordinate ruotato di 45° rispetto alle dimensioni del drone. Però, poiché solo due motori agiscono invece che quattro, l'attuazione dei comandi sarà più imprecisa e meno robusta rispetto al rumore, rendendo questa convenzione meno appetibile.

È possibile ottenere le relazioni tra i comandi sopra elencati e le forze (thrust – T) dei singoli motori (Figura 3), riportate di seguito:

$$\begin{bmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ -l & -l & l & l \\ l & -l & -l & l \\ d & -d & d & -d \end{bmatrix} \begin{bmatrix} T_1 \\ T_2 \\ T_3 \\ T_4 \end{bmatrix},$$

dove l è metà della lunghezza motore-motore lungo il lato, e d è il rapporto tra thrust e momento angolare.

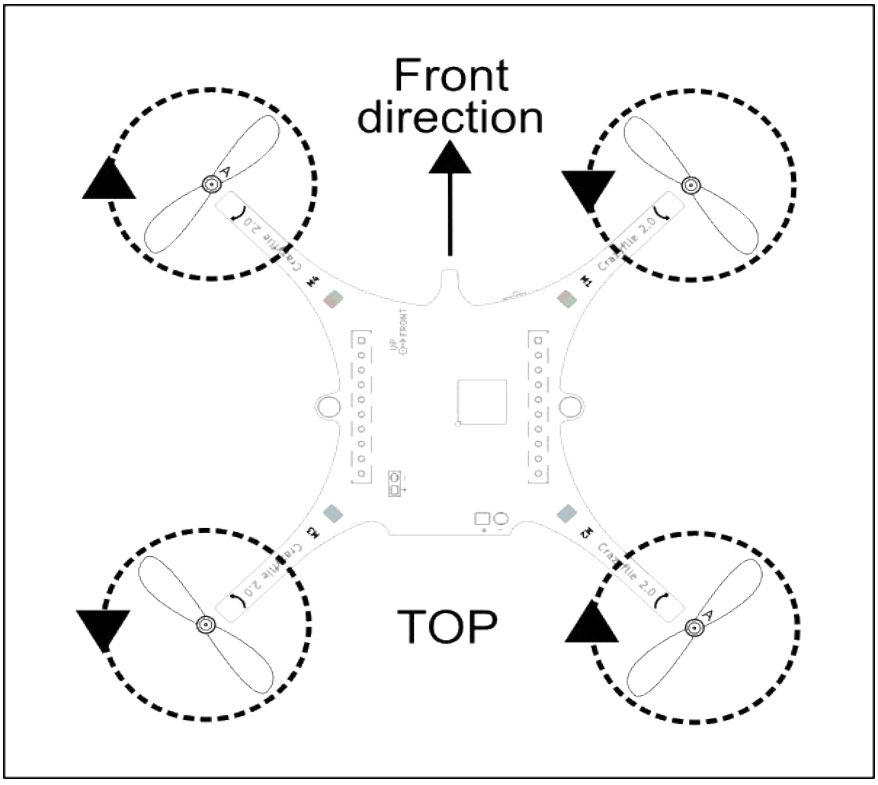

Figura 3 - Enumerazione dei motori e identificazione delle coppie oraria e antioraria

Essendo thrust e velocità di rotazione del motore (Ω) legati da una relazione di proporzionalità diretta (con coefficiente b):

$$T_i = b \cdot \Omega_i^2; \;\; \Omega_i \geqslant 0,$$

possiamo riscrivere l'equazione degli ingressi come:

$$\begin{bmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \end{bmatrix} = \begin{bmatrix} b & b & b & b \\ -lb & -lb & lb & lb \\ lb & -lb & -lb & lb \\ c & -c & c & -c \end{bmatrix} \begin{bmatrix} \Omega_1^2 \\ \Omega_2^2 \\ \Omega_3^2 \\ \Omega_4^2 \end{bmatrix}$$

dove $\mathbf{c}$ rappresenta il coefficiente di drag (ottenibile sperimentalmente, o come prodotto di $d \cdot b$) viii.

Inoltre le velocità di rotazione sono proporzionali alla tensione imposta sui motori, che a sua volta dipende dai valori di PWM (Σ); questi si possono decomporre in una componente di hover o trim (cioè che mantiene il drone stazionario), e una componente di deviazione da questo stato. Ovvero:

$$\begin{bmatrix} \Sigma_1 \\ \Sigma_2 \\ \Sigma_3 \\ \Sigma_4 \end{bmatrix} = \begin{bmatrix} \Sigma_{trim} \\ \Sigma_{trim} \\ \Sigma_{trim} \\ \Sigma_{trim} \end{bmatrix} + \begin{bmatrix} 1 & 1 & 1 & 1 \\ -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \begin{bmatrix} \Sigma_z \\ \Sigma_\phi \\ \Sigma_\theta \\ \Sigma_\psi \end{bmatrix},$$

dove il vettore di destra indica (in ordine) il comando di deviazione rispetto a spinta, rollio, beccheggio e imbardata.

2.3 - MODELLO DINAMICO

Il vettore di variabili di stato si compone di posizione $(x,\ y,\ z)$, velocità $(v_x,\ v_y,\ v_z)$, orientamento $(\phi,\ \theta,\ \psi)$ e variazione degli angoli di Eulero $(\dot{\phi},\ \dot{\theta},\ \dot{\psi})$, tutti espressi nel sistema di riferimento globale. Per avere una descrizione delle derivate degli stati (e quindi un modello del sistema dinamico), è necessario avere le equazioni cinematiche e dinamiche del quadrirotore.

Le relazioni tra le letture sensoriali, che sono espresse nel sistema di riferimento locale, e la loro controparte globale, sono le seguenti equazioni cinematiche:

$$\dot{\Gamma}^I = R \cdot V^B$$

$$\dot{\Theta}^I = T \cdot \omega^B,$$

dove Γ^I e Θ^I rappresentano la posizione e l'orientamento nel sistema di riferimento inerziale, mentre V^B e ω^B, le velocità lineari e angolari nel sistema di riferimento del drone (body-fixed); R e T sono la matrice di rotazione e la matrice di trasformazione (anche nota come l'inversa della matrice delle variazioni degli angoli di Eulero coniugata).ix

Le matrici di rotazione e trasformazione sono definite sotto, avendo avuto cura di rispettare l'inversione dell'angolo di pitch come previsto nella dichiarazione del sistema di riferimento:

$$R = \begin{bmatrix} c_\theta c_\psi & c_\theta s_\psi & s_\theta \\ -c_\phi s_\psi - s_\phi s_\theta c_\psi & c_\phi c_\psi - s_\phi s_\theta s_\psi & s_\phi c_\theta \\ s_\phi s_\psi - c_\phi s_\theta c_\psi & -s_\phi c_\psi - c_\phi s_\theta s_\psi & c_\phi c_\theta \end{bmatrix}$$

$$T = \frac{1}{c_\theta} * \begin{bmatrix} c_\theta & -s_\phi s_\theta & -c_\phi s_\theta \\ 0 & c_\phi c_\theta & -s_\phi c_\theta \\ 0 & s_\phi & c_\phi \end{bmatrix},$$

dove c_x e s_x sono abbreviazioni per $\sin(x)$ e $\cos(x)$.

Dalla seconda equazione di Newton si possono dedurre le espressioni delle forze (F) e dei momenti (τ):

$$F = \frac{d(mV)}{dt}$$

$$\tau = \frac{d(I\omega)}{dt}.$$

Dunque:

$$F^B = m\dot{V}^B + \omega^B \times mV^B$$

$$\tau^B = I\dot{\omega}^B + \omega^B \times I\omega^B,$$

dove la massa m e il momento d'inerzia I vengono dimensionate sperimentalmente.x

Queste equazioni possono essere correlate con gli ingressi descritti precedentemente:

$$F^B = R^{-1}\begin{bmatrix} 0 \\ 0 \\ -mg \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ U_1 \end{bmatrix}$$

$$\tau^B = \begin{bmatrix} U_2 \\ U_3 \\ U_4 \end{bmatrix}$$

Capitolo 3 - Struttura dell'anello di controllo

Il problema di controllo si può spezzare in due anelli in cascata: il primo serve per generare riferimenti di angolo e spinta partendo da coordinate di posizione e velocità (che possono essere statiche, provenire da una traiettoria, o essere decise interattivamente dall'utente); il secondo si occupa di attuare i riferimenti prodotti controllando direttamente i motori.xi Poiché l'ultimo anello ha tipicamente una frequenza di aggiornamento più alta del primo, viene chiamato interno (o anche di orientamento), e l'altro esterno (o anche di navigazione).

Si propone per prima la descrizione dell'anello di controllo interno, e successivamente l'esterno (Figura 4).

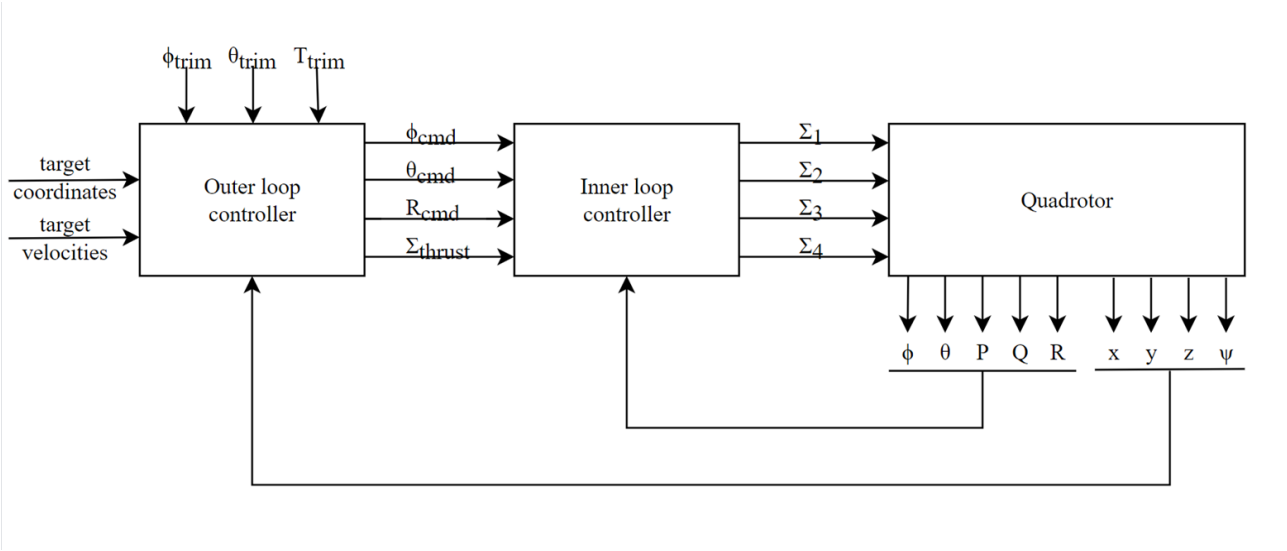

Figura 4 - Disposizione in cascata dei controllori

3.1 - ANELLO INTERNO

Dati gli ingressi U ci interessa poterli attuare comandando i singoli motori; questo può essere fatto interagendo con il firmware dal Crazyflie, che ha un parametro che accetta valori da 0 a 65535 (unsigned int a 16 bit), rappresentativi della PWM.

Un approccio potrebbe consistere nel generare gli ingressi (U) del sistema partendo dall'orientamento e spinta desiderati, per poi trasformare gli ingressi nelle forze che ogni motore deve esercitare, da lì passare alla velocità angolare, e infine alla PWM. Sfortunatamente la relazione che lega la PWM con la forza è quadratica, e richiede di conoscerne con precisione la funzione di trasferimento tra le due, che è dipendente da molti fattori che variano tra diversi motori ed eliche.[xii]

Un altro metodo consiste nel partire dalla quaterna di variabili di controllo da raggiungere (angoli di Eulero e spinta), e ottenere direttamente i comandi in PWM. Questo compito viene dato in carico a due ordini di controllori PID in cascata: il primo usa l'errore sull'orientamento per generare tre velocità angolari di correzione, mentre il secondo usa l'errore sulle velocità angolari per generare i comandi numerici (che corrispondono comunque agli ingressi U desiderati). Supponendo che la spinta sia già stata fornita nello stesso formato, è sufficiente sommare algebricamente i contributi per ogni motore, e poi inviarli al drone.

Questa è la soluzione implementata nel firmware del Crazyflie[xiii], ed è la soluzione che ho scelto di implementare essendo più diretta. Una modifica aggiuntiva consiste nel saltare il controllore sullo yaw nel primo ordine di PID, e di passare direttamente alla velocità angolare nel secondo livello; questa scelta è giustificata dal fatto che la misura di yaw è particolarmente affetta da errore di deriva, che si ripercuoterebbe sul resto del processo.

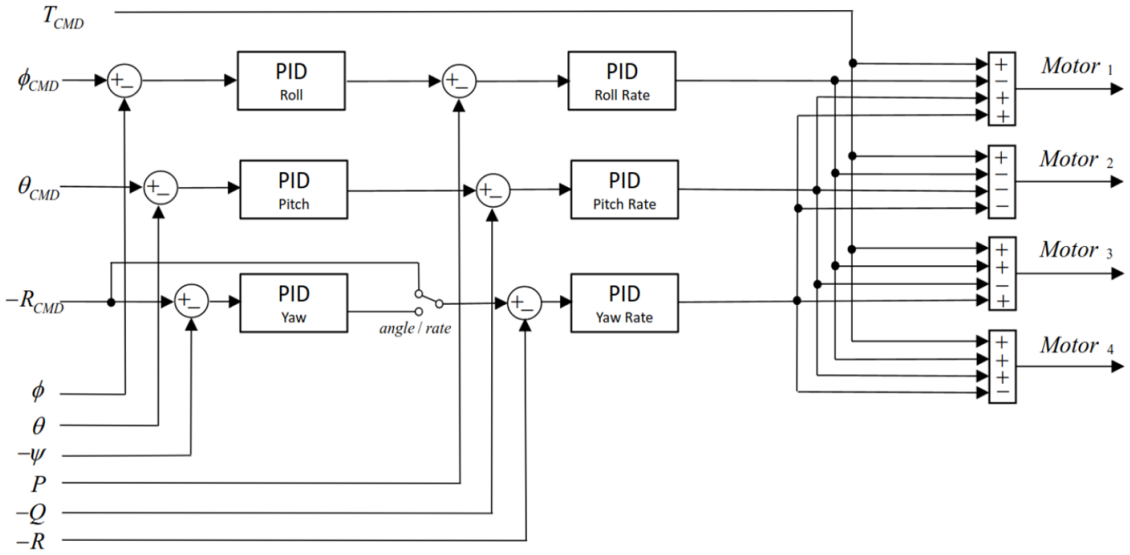

Figura 5 - Struttura dell'anello interno

In Figura 5 vediamo il diagramma dell'anello interno: con T_{CMD}, ϕ_{CMD}, θ_{CMD} e R_{CMD} si intendono il thrust e gli angoli di riferimento generati dall'anello di navigazione; da notare che R rappresenta lo yaw rate, e infatti la misura e il PID di imbardata non sono utilizzati. Le letture sensoriali sono indicate come ϕ, θ, ψ per gli angoli e P, Q, R per le velocità angolari. Infine $Motor_i$ rappresenta il comando in PWM relativo all' i-esimo motore.

3.2 - DESCRIZIONE CONTROLLORE PID

È utile a questo punto fornire una descrizione della struttura PID, che rappresenta un metodo di controllo molto popolare per la sua semplicità concettuale. Si compone di tre addendi: uno proporzionale, uno integrale e uno derivativo, da cui deriva l'acronimo. Queste operazioni sono relazionate all'errore di una misura rispetto ad un riferimento desiderato: dunque una combinazione lineare dell'errore, della sua derivata e del suo integrale (Figura 6).

In particolare l'azione proporzionale lega in modo direttamente proporzionale l'errore con il comando generato, e stabilisce l'intensità della risposta. Invece il termine integrale tiene conto dell'errore accumulatosi nel tempo: questo consente di superare eventuali equilibri che si potrebbero generare tra la sola azione proporzionale e l'ambiente, ad esempio con una forza elastica esterna dipendente dall'errore. Infine l'azione derivativa lavora sulla variazione dell'errore, e quindi influisce sulla reattività (o morbidezza) della risposta ed è necessaria per reagire con prontezza alle dinamiche dell'errore.

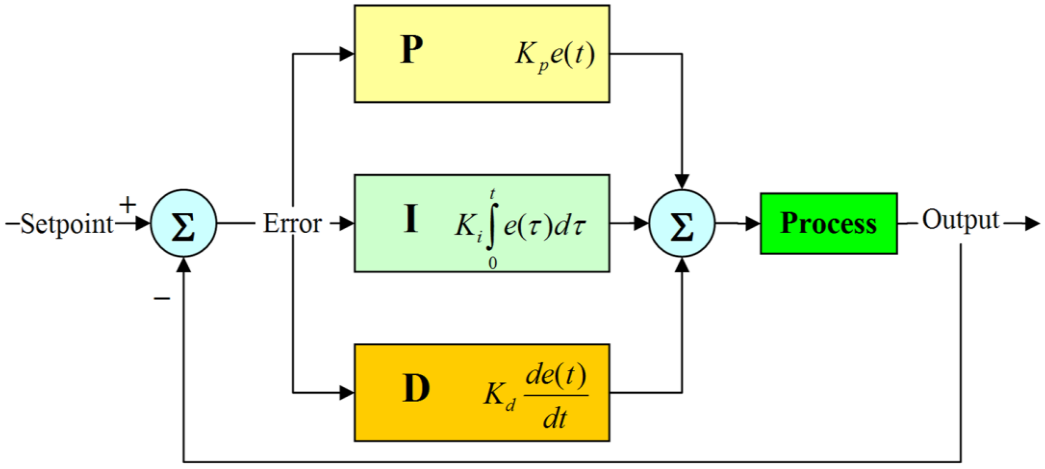

Figura 6 - Struttura controllore PID

Ogni operando è moltiplicato per un coefficiente che serve per settare l'intensità di ogni azione, e anche per stabilire il peso di un'azione rispetto alle altre. Poiché per tarare un PID è sufficiente calibrarne i coefficienti (spesso si procede per tentativi), il metodo è immediatamente applicabile a sistemi SISO (single input – single output); in sistemi MIMO (multiple input – multiple output), come questo è il caso, occorre prima scomporre il

problema in componenti ciascuna gestibile da un singolo PID (ad esempio attraverso disaccoppiamento).

La versione di PID che è stata implementato può essere consultata nell'Addendum. Ha delle funzionalità aggiuntive quali la possibilità di limitare il contributo integrale e l'uscita del controllore, l'aggiunta di un polo regolabile sull'azione derivativa per ottenere la reiezione di disturbi ad alta frequenza, e la derivata viene calcolata direttamente sulla misura invece che sull'errore (in quanto hanno la stessa dinamica ma si evitano salti bruschi in caso di discontinuità del riferimento)xiv.

3.3 - ANELLO ESTERNO

L'anello di controllo esterno serve a generare riferimenti di angolo e spinta a partire dalla posizione attuale del drone e i riferimenti forniti in posizione e velocità cartesiane. In particolare, si definisce un punto di lavoro in cui il drone si considera fermo, sul quale si sommano i comandi relativi alle deviazioni da questo stato. La cosa più conveniente è prendere lo stato di hover, a cui sono associati velocità, roll e pitch nulli, yaw rate arbitrario e thrust uguale al valore di PWM che non produce variazioni di quota (36000 dall'applicazione client). Questi valori sono anche detti di trim in quanto regolano la posizione del drone, sommandoli alle deviazioni.

Il calcolo della deviazione della spinta richiede solo di applicare la struttura PID alla differenza in elevazione tra il drone e il target, avendo cura che il comando sia dimensionato in PWM. Similmente, è possibile calcolare il comando di velocità angolare di imbardata dalla differenza di orientamento tra lo yaw del drone, e quello ottenibile dal vettore velocità di riferimento (attraverso l'arcotangente applicata sulle componenti orizzontali).

I delta su rollio e beccheggio solo legati da rispettivi PID alla distanza orizzontale tra drone e punto di inseguimento, ma queste distanze devono essere calcolate nel sistema di riferimento del drone, non in coordinate globali. Dunque attraverso l'applicazione della debita matrice di rotazione, si calcolano le distanze lungo la direzione indicata dallo yaw (chiamata along), e perpendicolare ad essa (across). In questo calcolo si suppone che il drone sia allineato con l'orientamento del riferimento; nei casi in cui la differenza di angolo sia significativa, le distanze verranno comunque ridotte a zero perché il PID di imbardata ne impone la convergenza, ma la traiettoria di avvicinamento sarà curvilinea.

$$S_{along} = x_{err} \cdot cos(\psi) + y_{err} \cdot sin(\psi)$$
$$S_{across} = x_{err} \cdot sin(\psi) - y_{err} \cdot cos(\psi)$$
$$\Delta\phi_{CMD}(t) = Kp_{along} \cdot S_{along}(t) + Ki_{along} \int S_{along}(t)\, dt$$
$$\Delta\theta_{CMD}(t) = Kp_{across} \cdot S_{across}(t) + Ki_{across} \int S_{across}(t)\, dt$$
$$\Delta R_{CMD}(t) = Kp_{\psi} \cdot \psi_{err}(t)$$
$$\Delta T_{CMD}(t) = Kp_{z} \cdot z_{err}(t) + Ki_{z} \int z_{err}(t)\, dt$$

Le equazioni sopra riportate riassumono l'algoritmo di navigazione: con x_{err}, y_{err}, z_{err}, ψ_{err} si indicano la distanza tra drone e target lungo gli assi del sistema di riferimento globale, e l'errore di imbardata; S_{along} e S_{across} sono la rappresentazione della stessa distanza nel sistema di riferimento locale (rispettivamente lungo il drone – x – e perpendicolarmente ad esso – y); $\Delta\phi_{CMD}$, $\Delta\theta_{CMD}$, ΔR_{CMD}, ΔT_{CMD} è invece la quaterna di comandi in uscita, su cui vanno sommati i rispettivi valori di trim.

3.4 - DIMENSIONAMENTO DEI PARAMETRI

I coefficienti dei PID sono stati calibrati a mano, partendo dai valori utilizzati nel sistema di controllo integrato nel drone. L'altro parametro fondamentale è la durata del ciclo (step time), che dovrà coincidere con la frequenza di aggiornamento delle letture sensoriali ricevute.

Capitolo 4 - Sistema di comunicazione con la piattaforma

Completata la progettazione della struttura di controllo si può passare all'implementazione pratica. È di cruciale importanza poter dialogare rapidamente con il drone, sia ricevendo le misure rilevate dai sensori, sia inviando i parametri di controllo a ciascun motore.

Come già menzionato la comunicazione con il pc è veicolata in radiofrequenza da Crazyradio, e governata attraverso l'API (cflib). Si usano quindi funzioni di alto livello, in realtà fondate su CRTP (Crazy RealTime Protocol), che mascherano le complessità della struttura di comunicazione, le modalità di generazione e le dinamiche dei singoli pacchetti.xv Non a caso occorre inizializzare i driver usando l'omonima libreria per rendere il drone operativo.xvi Da notare che essendo l'intero progetto open-source, è possibile customizzare tutti i livelli del modello di astrazione, ma ciò non è di interesse in questo progetto.

La comunicazione può avvenire a intervalli multipli di 10 msxvii, ma potenzialmente la frequenza è aumentabile modificando il firmware. L'iniziativa è sempre del computer in quanto il Crazyflie è programmato per inviare dati solo in risposta a richieste esterne. Per avere un flusso sensoriale continuo vengono inviati pacchetti eventualmente nulli (facendo polling).

Per iniziare la connessione con il drone si deve creare nel codice un'istanza di un oggetto Crazyflie, o SyncCrazyflie, specificandone l'URI (Uniform Resource Identifier), che contiene le informazioni di canale, banda e indirizzoxviii. È possibile ottenere i valori di default invocando le funzioni dalla libreria uri_helperxix, oppure ricavando questi valori

dall'applicazione cfclientxx, oppure ancora con una scansione di tutti gli URI rilevabili da Crazyradio.

Le operazioni di lettura/scrittura sul drone possono essere eseguite (invocando le relative funzioni in Python) solo su specifiche variabili, di cui è necessario conoscere il nome, il gruppo di appartenenza e il formato. La lista delle variabili, e le relative informazioni, sono specificate nelle tavole dei contenuti (ToC – Table of Contents), di cui ne esiste una per il loggingxxi, e una per i parametrixxii. La differenza tra i due è che i log sono in sola lettura dal drone, e contengono tipicamente le rilevazioni dei sensori (grezze o filtrate), o gli stati dei controllori integrati; invece i parametri possono essere di sola lettura, o consentire sia lettura che scrittura, e servono per controllare vari aspetti operativi del drone (ad esempio la scelta di quale controllore utilizzare o calibrarne i pesi).

In particolare per ricevere i log è necessario elencarli in un oggetto chiamato LogConfig e inviarlo preventivamente al drone, avendo cura di rispettare le dimensioni del pacchetto (26 Bytesxxiii).

La configurazione in uso nel progetto include le letture filtrate di posizione, velocità lineare, velocità angolare, e orientamento (espresso come quaternione compresso) che si trovano nel gruppo di log stateEstimateZ. Il peso complessivo è di 22 Byte, abbastanza per aggiungere anche una variabile di monitoraggio della batteria.

Inviato il LogConfig al drone, e impostata la frequenza di aggiornamento, ci si aspetta un flusso di pacchetti ininterrotto; per fare operazioni sui dati in arrivo si possono aggiungere funzioni di callback (ad esempio di scrittura su variabile), che verranno automaticamente chiamate ad ogni aggiornamento ricevuto.

Risolto il problema della ricezione, si passa ora al problema dell'attuazione dei motori: sarà sufficiente azionare il parametro che sgancia il controllo dei motori dalla gestione del firmware, abilitando la lettura in PWM da altri quattro parametri che si trovano nello stesso

gruppo. Il governo dei motori si realizza aggiornando ad ogni ciclo di controllo queste variabili, sempre rispettando la frequenza di comunicazione scelta.

Per garantire l'interattività con l'utente (per la gestione estemporanea dei punti di inseguimento, e l'arresto di sicurezza dei motori), è stato implementato un sistema su base threading. Ovvero il programma fin qui descritto rappresenta il thread principale, che viene unito a un secondo thread di lettura input. Questo è necessario per evitare che la lettura da tastiera risulti bloccante, caratteristica inaccettabile nel ciclo di controllo. I due thread dialogano attraverso una coda di lettura condivisa che opera con sole operazioni atomiche, rendendo sicuro lo scambio di dati.

Conclusioni

Il percorso identificato e descritto è solo uno dei possibili approcci attualmente disponibili in letteratura. Ad esempio altre possibilità prevedono l'integrazione di sistemi di motion capture, oppure l'utilizzo di diverse strutture di retroazione (ad esempio controllori LQR), diverse assunzioni nella creazione del modello dinamico, come anche algoritmi di navigazione, o addirittura prevedono di operare su espressioni di orientamento in quaternioni invece che in angoli di Eulero.

Questo studio infatti non vuole essere una discussione esaustiva sull'argomento, ma un'indagine su una possibilità di risoluzione del problema di stabilizzazione, nell'ottica speranzosa di un futuro utilizzo come riferimento per successivi lavori.

Appendice

In questo capitolo è riportato il codice Python che rispecchia nella pratica il percorso illustrato. Per il corretto funzionamento del programma è necessario installare l'interprete Python, assicurandosi la presenza delle librerie standard *math*, *time*, *queue* e *threading*, oltre a *readchar* per la lettura non bloccante di input da tastiera, e *cflib* per le operazioni riguardanti il drone.
Il programma è diviso in due files: *PID.py*, che contiene l'implementazione dell'omonimo controllore, e *control.py*, che gestisce lo scambio dati con Crazyflie e l'anello di controllo.

A1 - PID.py

```
import math


"""
  r  +   e   +-----+  u   +------------+      y
 ---> O ---> | PID | ---> | Attuators | ------->
    - ^      +-----+      +------------+     |
      | x                                    |
      |    +------------+     +----------+    |
      '--- | Filtering | <-- | Sensors | <--'
           +------------+     +----------+
"""


#u(t) = Kp * e(t) + Ki * ∫e(t)*dt + KD * d e(t)/dt * e^(-t/τ)/τ
#e(t) = r(t) - y(t)

#U(s)/E(s) = Kp  + Ki * 1/s + Kd * s/sτ+1

#s = 2/T * (z-1)/(z+1)

#u[n] = p[n] + i[n] + d[n]
#p[n] = Kp * e[n]
#i[n] = Ki*T/2 * (e[n] + e[n-1]) + i[n-1]
#d[n] = 2*Kd/(2τ+T) * (e[n] + e[n-1]) + (2τ-T)/(2τ+T) * d[n-1]


MAX_UINT16 = 65535 #maximum unsigned integer (used for PWM/motor control)


#NOTE: deprecated!
#values are taken from the crazyflie default configuration (cfclient -> tuning)
class DefaultPIDparams:
    def __init__(self, T):
```

```
        self.x = {"Kp":5, "Ki":5, "Kd":5, "T":T, "limMin":0., "lim-
Max":MAX_UINT16, "limMinInt":-MAX_UINT16, "limMaxInt":MAX_UINT16}
        self.y = {"Kp":5, "Ki":5, "Kd":5, "T":T, "limMin":0., "lim-
Max":MAX_UINT16, "limMinInt":-MAX_UINT16, "limMaxInt":MAX_UINT16}
        self.z = {"Kp":5, "Ki":5, "Kd":5, "T":T, "limMin":0., "lim-
Max":MAX_UINT16, "limMinInt":-MAX_UINT16, "limMaxInt":MAX_UINT16}
        self.roll = {"Kp":5, "Ki":5, "Kd":5, "T":T, "limMin":0., "lim-
Max":MAX_UINT16, "limMinInt":-MAX_UINT16, "limMaxInt":MAX_UINT16}
        self.pitch = {"Kp":5, "Ki":5, "Kd":5, "T":T, "limMin":0., "lim-
Max":MAX_UINT16, "limMinInt":-MAX_UINT16, "limMaxInt":MAX_UINT16}
        self.yaw = {"Kp":5, "Ki":5, "Kd":5, "T":T, "limMin":0., "lim-
Max":MAX_UINT16, "limMinInt":-MAX_UINT16, "limMaxInt":MAX_UINT16}
        self.vx = {"Kp":25, "Ki":25, "Kd":5, "T":T, "limMin":0., "lim-
Max":MAX_UINT16, "limMinInt":-MAX_UINT16, "limMaxInt":MAX_UINT16}
        self.vy = {"Kp":25, "Ki":25, "Kd":5, "T":T, "limMin":0., "lim-
Max":MAX_UINT16, "limMinInt":-MAX_UINT16, "limMaxInt":MAX_UINT16}
        self.vz = {"Kp":25, "Ki":25, "Kd":5, "T":T, "limMin":0., "lim-
Max":MAX_UINT16, "limMinInt":-MAX_UINT16, "limMaxInt":MAX_UINT16}
        self.rollRate = {"Kp":500, "Ki":500, "Kd":5, "T":T, "limMin":0.,
"limMax":MAX_UINT16, "limMinInt":-MAX_UINT16, "limMaxInt":MAX_UINT16}
        self.pitchRate = {"Kp":500, "Ki":500, "Kd":5, "T":T, "limMin":0.,
"limMax":MAX_UINT16, "limMinInt":-MAX_UINT16, "limMaxInt":MAX_UINT16}
        self.yawRate = {"Kp":100, "Ki":50, "Kd":5, "T":T, "limMin":0., "lim-
Max":MAX_UINT16, "limMinInt":-MAX_UINT16, "limMaxInt":MAX_UINT16}



class PID:
    def __init__(self, Kp=10, Ki=2, Kd=1, T=1, tau=0.001,
                 limMin = 0, limMax = math.inf,
                 limMinInt = -math.inf, limMaxInt = math.inf, tolerance=0):

        #control coefficients
        self.Kp = Kp
        self.Ki = Ki
        self.Kd = Kd

        #step time
        self.T = T #should be less than System_BW/10

        #low-pass derivative filter to reject HF noise
        self.tau = tau

        #output clamping
        self.limMin = limMin
        self.limMax = limMax

        #integral clamping
        self.limMinInt = limMinInt
        self.limMaxInt = limMaxInt

        #integrator
        self.integrator = 0
        self.prevError = 0 #for integrator
```

```
        #differentiator; derivative of Input instead of Error to avoid jumps
when the Target changes
        self.differentiator = 0
        self.prevInput = 0 #for differentiator

        #for angle re-wrapping
        self.tolerance = tolerance
        self.prevTarget = 0
        self.flip_counter = 0
        self.target_flip_counter = 0
        self.input_flip_counter = 0

        #saved just for external ease of access
        self.error = 0
        self.output = 0

    def update (self, input, target):
        #If input or target represent angles, +/- 180° is a critical angle,
        #since, if increased, the angle will "flip" with a 360° excursion.
        #To avoid this behaviour we must compensate with a counter-rotation,
times the number of flips.

        #if tolerance is set to 0 skip the above considerations
        if self.tolerance != 0:
            #count number of flips, if any
            if math.abs(target-self.prevTarget) > self.tolerance:
                if target > self.prevTarget:
                    self.target_flip_counter += 1
                else:
                    self.target_flip_counter -= 1

            if math.abs(input-self.prevInput) > self.tolerance:
                if input > self.prevInput:
                    self.input_flip_counter += 1
                else:
                    self.input_flip_counter -= 1

            #compensate the flips
            target = target - self.target_flip_counter * 360
            input = input - self.input_flip_counter * 360



        #error
        self.error = target - input

        #proportional
        proportional = self.Kp * self.error

        #integral
        self.integrator = self.integrator + self.Ki * self.T * (self.error +
self.prevError)/2

        #anti-wind-up using integrator clamping
        if (self.integrator > self.limMaxInt):
            self.integrator = self.limMaxInt
```

```
        elif (self.integrator < self.limMinInt):
            self.integrator = self.limMinInt

        #differentiator
        self.differentiator = - (2 * self.Kd * (input - self.prevInput) + (2
* self.tau - self.T) * self.differentiator) / (2 * self.tau + self.T)

        #output
        self.output = proportional + self.integrator + self.differentiator

        if (self.output > self.limMax):
            self.output = self.limMax
        elif (self.output < self.limMin):
            self.output = self.limMin

        #store error, input & target for PID.update() call
        self.prevError = self.error
        self.prevInput = input
        self.prevTarget = target

        return self.output
```

A2 - control.py

```
import time
import math

import queue
import threading
import readchar

import cflib.crtp
from cflib.utils import uri_helper
from cflib.crazyflie import Crazyflie
from cflib.crazyflie.syncCrazyflie import SyncCrazyflie
from cflib.crazyflie.log import LogConfig

from PID import PID

"""
DUE PAROLE SUI SISTEMI DI RIFERIMENTO
Il crayzyflie utilizza un sistema di riferimento fissato sul corpo rigido di
tipo ENU
(ovvero East-North-Up, che identifica in quest'ordine la direzione positiva
degli assi x-y-z).
Il fronte del drone può essere identificato dalla ricetrasmittente che
sporge dal frame,
mentre il retro può essere identificato dai due led di colore blu.
Visto dall'alto il sistema di riferimento è:

        ^  asse x
        |
        |
  < ----o  asse z (uscente)
  asse y
```

```
Inoltre gli angoli sono così definiti:
- roll: (φ - phi) come rotazione intorno all'asse x (positiva in senso ora-
rio) -> positivo quando inclina verso destra
- pitch: (θ - theta) come rotazione intorno all'asse y (positiva in senso
antiorario) -> positivo quando il drone si impenna
- yaw: (ψ - psi) come rotazione intorno all'asse z (positiva in senso ora-
rio) - positivo virando verso sinistra

In angoli di Eulero, questo ricorda lo standard 1-2-3 (ordine di rotazione
degli assi)
"""

URI = uri_helper.uri_from_env()

AUTOMATIC_CONTROL = 0
MANUAL_CONTROL = 1 #bool

MAX_UINT16 = 65535 #maximum unsigned integer (used for PWM/motor control)
HOVER_THRUST = 36000 #default value to obtain stationary behaviour in alti-
tude (32768/48000)

MOTOR_MAX_FORCE = 0.15 #N, max force that can be exherted by 1 motor
MOTOR_MAX_TORQUE = 0.91e-3 #N*m, torque outputted at max force by 1 motor
MAX_FLIGHT_TIME = 7 #in minutes
DEFAULT_HEIGHT = 1 #in meters

T = 0.01 #s, step time, must be >= CF logging rate (0.01) + the cycle compu-
tation time

DRONE_DIAGONAL_LENGHT = 0.092 #in m; distance between opposite motors
DRONE_SIDE_LENGHT = 0.092/math.sqrt(2) #in m; distance between adiacent mo-
tors
DRONE_MASS = 28.0e-3 #Kg, +/- 0.1 grams
DRONE_FORCE_TO_TORQUE = 0.00596 #proportional factor, between the force of a
motor and the torque it generates

user_input = queue.Queue(1) #a queue with maxsize=1 is used to ensure that
operations between threads are atomic


class Sensor_logging:
    battery_level = 0

    origin = [0, 0, 0] #position offset (in m) so that (x, y, z) are rela-
tive to it. Set with set_origin()
    x, y, z = 0, 0, 0 #position in mm
    vx, vy, vz = 0, 0, 0 #velocity in mm/s
    #ax, ay, az = 0, 0, 0 #acceleration in mm/s^2, Z includes gravity

    qw, qx, qy, qz = 0, 0, 0, 0 #attitude as a quaternion, unpacked from raw
packet sent by logging
    roll, pitch, yaw = 0, 0, 0 #euler angles in degrees

    p, q, r = 0, 0, 0 #angular velocity in millirad/s
```

```
    roll_rate, pitch_rate, yaw_rate = 0, 0, 0 #euler angle rates in milli-
rad/s

    def set_origin(self): #ONLY CALL ONCE!
        self.origin[0] = self.x
        self.origin[1] = self.y
        self.origin[2] = self.z

    ### CALLBACKS ###
    def battery_log_cb (self, timestamp, data, logconf):
        self.battery_level=data["pm.batteryLevel"]

    def position_cb(self, timestamp, data, logconf):
        #the starting position (origin) of the crazyflie is subtracted from
x, y, z to provide the distance relative to it
        self.x = data["stateEstimateZ.x"] - self.origin[0]
        self.y = data["stateEstimateZ.y"] - self.origin[1]
        self.z = data["stateEstimateZ.z"] - self.origin[2]

    def velocity_cb(self, timestamp, data, logconf):
        self.vx = data["stateEstimateZ.vx"]
        self.vy = data["stateEstimateZ.vy"]
        self.vz = data["stateEstimateZ.vz"]

    """def acceleration_cb (self, timestamp, data, logconf):
        self.ax = data["stateEstimateZ.ax"]
        self.ay = data["stateEstimateZ.ay"]
        self.az = data["stateEstimateZ.az"]"""

    def attitude_cb(self, timestamp, data, logconf):
        #unpacking algorithm from the crazyflie firmware, original in  C
(modules/interface/quatcompress.h)
        comp = data["stateEstimateZ.quat"]
        mask = (1 << 9) - 1
        i_largest = comp >> 30
        sum_squares = 0
        q = [0, 0, 0, 0]

        #unpacking the quaternion
        for i in range(3, -1, -1): #3, 2, 1, 0
            if (i != i_largest):
                mag = comp & mask
                negbit = (comp >> 9 ) & 0x1
                comp = comp >> 10
                q[i] = 1/math.sqrt(2) * mag / mask
                if (negbit == 1):
                    q[i] = -q[i]
                sum_squares += q[i] * q[i]

        q[i_largest] = math.sqrt(1.0 - sum_squares)

        #update quaternion
        self.qx, self.qy, self.qz, self.qw = q

        #update attitude
        roll_r, pitch_r, yaw_r = euler_from_quaternion(*q)
```

```
        self.roll, self.pitch, self.yaw = math.degrees(roll_r), math.de-
grees(pitch_r), math.degrees(yaw_r)

    def angular_velocity_cb(self, timestamp, data, logconf):
        self.p = data["stateEstimateZ.rateRoll"]
        self.q = data["stateEstimateZ.ratePitch"]
        self.r = data["stateEstimateZ.rateYaw"]

        s_roll = math.sin(math.radians(self.roll))
        c_roll = math.cos(math.radians(self.roll))
        c_pitch = math.cos(math.radians(self.pitch))
        t_pitch = math.tan(math.radians(self.pitch))

        self.roll_rate = self.p - s_roll*t_pitch * self.q - c_roll*t_pitch *
self.r
        self.pitch_rate = c_roll * self.q - s_roll * self.r
        self.yaw_rate = s_roll/c_pitch * self.q + c_roll/c_pitch * self.r

    ### PRINT STATES ###
    def print_battery_level(self):
        print("BATTERY LEVEL: {0}% - {1} MINUTES OF EXPECTED FLIGHT
TIME".format(self.battery_level, self.battery_level*MAX_FLIGHT_TIME/100))
    def print_position(self):
        print("POSITION:\n X: {0:.1f} mm,     Y: {1:.1f} mm,     Z: {2:.1f}
mm".format(self.x, self.y, self.z))
    def print_velocity(self):
        print("VELOCITY:\n X: {0:.1f} mm/s,   Y: {1:.1f} mm/s,   Z: {2:.1f}
mm/s".format(self.vx, self.vy, self.vz))
    """def print_aceleration(self):
        print("ACCELERATION:\n X: {0:.4f} m/s^2, Y: {1:.4f} m/s^2, Z:
{2:.4f} m/s^2".format(self.ax/1000, self.ay/1000, self.az/1000))"""
    def print_attitude(self):
        print("ATTITUDE:\n ROLL: {0:.4f}°, PITCH: {1:.4f}°, YAW:
{2:.4f}°".format((self.roll),

(self.pitch),

(self.yaw)))
    def print_quaternion(self):
        print("QUATERNION:\n [{0:.4f}, {1:.4f}, {2:.4f}, {3:.4f}]".for-
mat(self.qw, self.qx, self.qy, self.qz))
    def print_rates(self):
        print("EULER RATES: \n ROLL: {0:.4f}°/s, PITCH: {1:.4f}°/S, YAW:
{2:.4f}°/s".format(math.degrees(self.roll_rate/1000),

math.degrees(self.pitch_rate/1000),

math.degrees(self.yaw_rate/1000)))
    def print_angular_velocity(self):
        print("A. VELOCITY:\n P: {0:.4f}°/s, Q: {1:.4f}°/S, R:
{2:.4f}°/s".format(math.degrees(self.p/1000),

math.degrees(self.q/1000),

math.degrees(self.r/1000)))
```

```
    def print_all(self):
        self.print_position()
        self.print_velocity()
        self.print_attitude()
        self.print_rates()

def clear_screen():
    print("\033[H\033[J", end="")

### MOTOR CONTROL ###
def set_motor_power (crazyflie, motor_n, *, power=0, percent=False):
    """Mask to streamline the motor PWM control
    @crazyflie is the SyncCrazyFlie instance,
    @motor_n is the selected motor (1 to 4)
    @power is the PWM value in uint_16 (from 0 to 65535), or float (0% to
100%)
    @percent is a bool used to select the @power mode"""

    if (motor_n < 1 or motor_n > 4):
        raise ValueError("Please select a motor number between 1 and 4")

    if not percent and type(power) != int:
        raise TypeError("Motor power must be expressed in integers when per-
cent flag is False")

    if not percent and (power<0 or power>MAX_UINT16):
        raise ValueError("Motor power must be set between 0 and {0}".for-
mat(MAX_UINT16))

    if percent and (power<0 or power>100):
        raise ValueError("Motor power must be set between 0%% and 100%")

    if percent:
        power = int(MAX_UINT16 * power / 100)

    crazyflie.cf.param.set_value("motorPowerSet.m" + str(motor_n), power)

def set_motor_control_mode(crazyflie, mode):
    """Mode must be either AUTOMATIC_CONTROL (0) or MANUAL_CONTROL (1)"""
    if mode != MANUAL_CONTROL and mode != AUTOMATIC_CONTROL:
        raise ValueError("Motor control mode must be either automatic or
manual")

    crazyflie.cf.param.set_value("motorPowerSet.enable", mode)

    if mode == MANUAL_CONTROL:
        print("Crazyflie motors have been set to manual control mode")
    elif mode == AUTOMATIC_CONTROL:
        print("Crazyflie motors have been set to automatic control mode")

def get_motor_control_mode(crazyflie):
    return crazyflie.cf.param.get_value("motorPowerSet.enable")

def stop_motors(crazyflie):
    set_motor_power(crazyflie=crazyflie, motor_n=1, power=0, percent=1)
    set_motor_power(crazyflie=crazyflie, motor_n=2, power=0, percent=1)
```

```
    set_motor_power(crazyflie=crazyflie, motor_n=3, power=0, percent=1)
    set_motor_power(crazyflie=crazyflie, motor_n=4, power=0, percent=1)
    print("All motor PWM values are now set to 0")


### CONVERSION ###
def euler_from_quaternion(x, y, z, w):
        """
        Convert a quaternion into euler angles (roll, pitch, yaw)
        roll is rotation around x in radians (counterclockwise)
        pitch is rotation around y in radians (clockwise)
        yaw is rotation around z in radians (counterclockwise)
        """
        t0 = +2.0 * (w * x + y * z)
        t1 = +1.0 - 2.0 * (x * x + y * y)
        roll_x = math.atan2(t0, t1)

        t2 = +2.0 * (w * y - z * x)
        t2 = +1.0 if t2 > +1.0 else t2
        t2 = -1.0 if t2 < -1.0 else t2
        pitch_y = - math.asin(t2)

        t3 = +2.0 * (w * z + x * y)
        t4 = +1.0 - 2.0 * (y * y + z * z)
        yaw_z = math.atan2(t3, t4)

        return roll_x, pitch_y, yaw_z # in radians


### TESTS ###

def motor_test(scf):

    stop_motors(scf)

    if get_motor_control_mode(scf) == AUTOMATIC_CONTROL:
        print("Motor control mode is currently automatic")
    else:
        print("Motor control mode is currently manual")

    set_motor_control_mode(scf, MANUAL_CONTROL)


    try:
        for power in range(0, 101, 5):
            set_motor_power(crazyflie=scf, motor_n=1, power=power, per-
cent=True)
            set_motor_power(crazyflie=scf, motor_n=2, power=power, per-
cent=True)
            set_motor_power(crazyflie=scf, motor_n=3, power=power, per-
cent=True)
            set_motor_power(crazyflie=scf, motor_n=4, power=power, per-
cent=True)
            print("Motor power: {0}%".format(power))
            time.sleep(1)
```

```
        stop_motors(scf)
        time.sleep(0.02)

        set_motor_control_mode(scf, AUTOMATIC_CONTROL)

    except KeyboardInterrupt("Program interrupted by user"):
        stop_motors(scf)
        set_motor_control_mode(scf, AUTOMATIC_CONTROL)


### THREADS ###
def read_user_input():
    global user_input
    while 1:
        if user_input.empty():
            key = readchar.readkey()
            user_input.put(key)

        #"0" button press must exit the program; all threads must quit when
pressed
        if key == "0":
            break


def main_program():
    cflib.crtp.init_drivers()
    sensors = Sensor_logging()


    with SyncCrazyflie(link_uri=URI, cf=Crazyflie(rw_cache="./cache")) as
scf:
        scf.wait_for_params()

        ##### CREATE LOGGING ENVIRONMENT #####
        # # DECLARE & LOAD LOGCONFIG CONTENTS # #
        drone_state_log = LogConfig(name="12states", period_in_ms=10)
        drone_state_log.add_variable(name="stateEstimateZ.x",
fetch_as="int16_t")
        drone_state_log.add_variable(name="stateEstimateZ.y",
fetch_as="int16_t")
        drone_state_log.add_variable(name="stateEstimateZ.z",
fetch_as="int16_t")

        drone_state_log.add_variable(name="stateEstimateZ.vx",
fetch_as="int16_t")
        drone_state_log.add_variable(name="stateEstimateZ.vy",
fetch_as="int16_t")
        drone_state_log.add_variable(name="stateEstimateZ.vz",
fetch_as="int16_t")

        #drone_state_log.add_variable(name="stateEstimateZ.ax",
fetch_as="int16_t")
        #drone_state_log.add_variable(name="stateEstimateZ.ay",
fetch_as="int16_t")
        #drone_state_log.add_variable(name="stateEstimateZ.az",
fetch_as="int16_t")
```

```
        drone_state_log.add_variable(name="stateEstimateZ.quat",
fetch_as="uint32_t")

        drone_state_log.add_variable(name="stateEstimateZ.rateRoll",
fetch_as="int16_t")
        drone_state_log.add_variable(name="stateEstimateZ.ratePitch",
fetch_as="int16_t")
        drone_state_log.add_variable(name="stateEstimateZ.rateYaw",
fetch_as="int16_t")

        drone_state_log.add_variable(name="pm.batteryLevel",
fetch_as="uint8_t")

        scf.cf.log.add_config(drone_state_log)

        # # ADD CALLBACKS # #
        drone_state_log.data_received_cb.add_callback(sensors.bat-
tery_log_cb)
        drone_state_log.data_received_cb.add_callback(sensors.position_cb)
        drone_state_log.data_received_cb.add_callback(sensors.velocity_cb)
        #drone_state_log.data_received_cb.add_callback(sensors.accelera-
tion_cb)
        drone_state_log.data_received_cb.add_callback(sensors.attitude_cb)
        drone_state_log.data_received_cb.add_callback(sensors.angular_veloc-
ity_cb)

        # # START LOGGING # #
        drone_state_log.start()
        time.sleep(0.1) #wait for the logging to actually start

        sensors.set_origin() #set the origin of the world frame as the
starting position of the Crazyflie
        time.sleep(0.1) #wait for the origin measurement to affects position



        ##### PRE-FLIGHT CHECKS #####
        # # MINIMUM BATTERY CHECK # #
        """if (sensors.battery_level <= 20):
            while (1):
                cmd = input("Battery level low ({0}%): do you wish to pro-
ceed anyways?\n[Y/N]: ".format(sensors.battery_level))
                if cmd == "Y" or cmd == "y":
                    break
                elif cmd == "N" or cmd == "n":
                    exit()
                else:
                    print ("Invalid input")"""

        # # SENSORS CHECK # #
        if (not sensors.x and not sensors.vx and not sensors.roll):
            print("Sensors measurements are dead!!")
            sensors.print_all()
            exit()
```

```
        # # SET MOTOR CONTROL MODE # #
        stop_motors(scf)

        if get_motor_control_mode(scf) == AUTOMATIC_CONTROL:
            print("Motor control mode is currently automatic")
        else:
            print("Motor control mode is currently manual")

        set_motor_control_mode(scf, MANUAL_CONTROL)



        ##### DECLARE LOOP VARIABLES #####
        # # SET TRIM (HOVER) VARIABLES # #
        thrust_trim = HOVER_THRUST
        roll_trim = 0
        pitch_trim = 0
        yaw_rate_trim = 0

        # # SET TARGET COORDINATES # #
        #this is used when following a pre-existing flight plan
        flight_plan_coordinates = [0, 0, 0] #mm
        flight_plan_velocity = [0, 0, 0] #mm #NOTE: seems to only be used
for yaw_CMD determination

        # # SET USER COORDINATES # #
        #this is used to let the user control the drone with the pc keyboard
        user_coordinates = [0, 0, 0] #mm
        user_yaw = [0] #in degrees #NOTE:only used if user control is active

        # # OTHER VARIABLES & FLAGS # #
        motor_PWM = [0, 0, 0, 0]

        pre_PID_time = 0
        outer_PID_time = 0
        inner_PID_time = 0
        update_motors_time = 0
        post_PID_time = 0
        cycle_time = 0

        global user_input
        user_control = False



        ##### PID INITIALIZATION #####
        #NOTE: tau parameter (for the derivative filter) is set to default
value
        MAX_ANGLE = math.radians(20) #clamp max angle to avoid divergence
from the linerized model
        MAX_RATE = math.radians(20) #clamp max angle rates to avoid sudden
movements

        # # EXTERNAL/COMMAND LOOP # #
        along_PID_params = {"Kp":0.2, "Ki":0, "Kd":0, "T":T}
        across_PID_params = {"Kp":0.2, "Ki":0, "Kd":0, "T":T}
```

```
        yaw_PID_params = {"Kp":6, "Ki":0, "Kd":0, "T":T, "tolerance":300}
        height_PID_params = {"Kp":2, "Ki":0.5, "Kd":0, "T":T}

        along_PID = PID(**along_PID_params)
        across_PID = PID(**across_PID_params)
        yaw_PID = PID(**yaw_PID_params)
        height_PID = PID(**height_PID_params)

        # # INTERNAL/ATTITUDE LOOP # #
        roll_PID_params = {"Kp":6, "Ki": 3, "Kd":0, "T":T, "tolerance":300}
        pitch_PID_params = {"Kp":6, "Ki": 3, "Kd":0, "T":T, "tolerance":300}
        roll_rate_PID_params = {"Kp":25, "Ki": 5, "Kd":2.5, "T":T}
        pitch_rate_PID_params = {"Kp":25, "Ki": 5, "Kd":2.5, "T":T}
        yaw_rate_PID_params = {"Kp":12, "Ki": 16.7, "Kd":0, "T":T}

        roll_PID = PID(**roll_PID_params)
        pitch_PID = PID(**pitch_PID_params)
        roll_rate_PID = PID(**roll_rate_PID_params)
        pitch_rate_PID = PID(**pitch_rate_PID_params)
        yaw_rate_PID = PID(**yaw_rate_PID_params)



        ##### CONTROL LOOP #####
        while 1:
            cycle_start = time.process_time()

            time_buoy = time.process_time() #time snapshot

            clear_screen()

            #unreliable since the voltage fluctuates dramatically during
flight
            """if sensors.battery_level <= 10:
                print("Battery is dangerously low, shutting motors")
                break"""


            ##### MANAGE USER INPUT #####
            if user_input.full():
                key = user_input.get()

                # # EXIT CONDITION # #
                if key == "0":
                    break

                # # START/STOP USER CONTROL # #
                if key == readchar.key.ENTER:
                    if user_control == False:
                        user_control = True
                        #when user takes control, preserve the state of the
drone to avoid sudden motions/crashes
                        user_coordinates = [target_x, target_y, target_z]
                        user_yaw = yaw_CMD
                    else:
                        user_control = False
```

```
                        #TODO what to do about target coordinates when user
deactivates manual control?

                # # UPDATE USER COORDINATES # #
                if user_control == True:
                #x axis (positive forward)
                    if key == "w":
                        user_coordinates[0] += 100 #add 10 cm forward
                    elif key == "s":
                        user_coordinates[0] -= 100 #add 10 cm back
                    #y axis (positive left)
                    elif key == "a":
                        user_coordinates[1] += 100 #add 10 cm left
                    elif key == "d":
                        user_coordinates[1] -= 100 #add 10 cm right
                    #z axis (positive up)
                    elif key == readchar.key.SPACE:
                        user_coordinates[2] += 100 #add 10 cm up
                    elif key == "c":
                        user_coordinates[2] -= 100 #add 10 cm down
                    #yaw (positive left)
                    elif key == "q":
                        user_yaw += 9 #add 9 degrees left
                    elif key == "e":
                        user_yaw -= 9 #add 9 degress right



            ##### MANAGE PIDS #####
            # # # COMPUTE TARGET AND DISTANCE FROM IT # # #
            if user_control == False:
                target_x, target_y, target_z = flight_plan_coordinates
                target_vx, target_vy, target_vz = flight_plan_velocity
                yaw_CMD = math.degrees(math.atan2(target_vy, target_vx))

            else:
                target_x, target_y, target_z = user_coordinates
                yaw_CMD = user_yaw

            #distance between drone and target in world coordinates
            delta_x = target_x - sensors.x
            delta_y = target_y - sensors.y
            #same distance expressed along and across the velocity of the
target point
            along = delta_x * math.cos(math.radians(yaw_CMD)) + delta_y *
math.sin(math.radians(yaw_CMD))
            across = delta_x * math.sin(math.radians(yaw_CMD)) - delta_y *
math.cos(math.radians(yaw_CMD))

            #time elapsed before PID computation
            pre_PID_time = time.process_time() - time_buoy
            time_buoy = time.process_time()
```

```
            # # # UPDATE COMMAND/OUTER PIDS # # #
            #computing the deviation from the hover condition
            roll_CMD = along_PID.update(input=along, target=0) #target is 0
because we want the distance to converge to 0
            pitch_CMD = across_PID.update(input=across, target=0)
            yaw_rate_CMD = yaw_PID.update(input=sensors.yaw, target=yaw_CMD)
            thrust_CMD = height_PID.update(input=sensors.z, target=target_z)

            #adding the hover state
            roll_CMD += roll_trim
            pitch_CMD += pitch_trim
            yaw_rate_CMD += yaw_rate_trim
            thrust_CMD = int(thrust_CMD + thrust_trim)

            #time elapsed to convert position into angles & thrust CMD
            outer_PID_time = time.process_time() - time_buoy
            time_buoy = time.process_time()



            # # # UPDATE ATTITUDE/INNER PIDS # # #
            roll_rate_CMD = roll_PID.update(input=sensors.roll, tar-
get=roll_CMD)
            pitch_rate_CMD = pitch_PID.update(input=sensors.pitch, tar-
get=pitch_CMD)

            #despite being called roll/pitch/yaw rates, they are actually
the angular velocities
            thrust_PWM = int(thrust_CMD)
            roll_PWM = int(roll_rate_PID.update(input=sensors.p, tar-
get=roll_rate_CMD))
            pitch_PWM = int(pitch_rate_PID.update(input=-sensors.q, tar-
get=pitch_rate_CMD))
            yaw_PWM = int(yaw_rate_PID.update(input=-sensors.r, target=-
yaw_rate_CMD))

            #time elapsed to convert angles & thrust commands into PWM com-
mands
            inner_PID_time = time.process_time() - time_buoy
            time_buoy = time.process_time()



            ##### SEND MOTOR PWM COMMANDS #####
            motor_PWM[0] = thrust_PWM - roll_PWM + pitch_PWM  + yaw_PWM
            motor_PWM[1] = thrust_PWM - roll_PWM - pitch_PWM  - yaw_PWM
            motor_PWM[2] = thrust_PWM + roll_PWM - pitch_PWM  + yaw_PWM
            motor_PWM[3] = thrust_PWM + roll_PWM + pitch_PWM  - yaw_PWM

            #clamp PWM to avoid saturation or dangerous behaviour at max
thrust
            for i in range (0,4):
                if motor_PWM[i] > MAX_UINT16: motor_PWM[i] = MAX_UINT16
                if motor_PWM[i] < 0: motor_PWM[i] = 0
```

```
            #attuate the PWM command
            #set_motor_power(scf, 1, power=motor_PWM[0])
            #set_motor_power(scf, 2, power=motor_PWM[1])
            #set_motor_power(scf, 3, power=motor_PWM[2])
            #set_motor_power(scf, 4, power=motor_PWM[3])

            #time elapsed to send the PWM command to the cf
            update_motors_time = time.process_time() - time_buoy
            time_buoy = time.process_time()



            if user_control == True:
                print("WARNING: user control is active, ENTER to deacti-
vate.")
            else:
                print("To activate manual control press ENTER.\nBe sure to
know the control scheme before doing so.")
            print("")

            sensors.print_all()
            print("")

            print("Target position: ({0:.1f}, {1:.1f}, {2:.1f}) mm; target
velocity: ({3:.1f}, {4:.1f}, {5:.1f}) mm/s"
                  .format(target_x, target_y, target_z, target_vx, tar-
get_vy, target_vz))

            print("Distance from target\nX: {0:.1f} mm, Y: {1:.1f} mm, Z:
{2:.4f} mm ({3:.1f} mm across and {4:.1f} mm along)"
                  .format(delta_x, delta_y, target_z-sensors.z, across,
along))
            print("")

            print("Outer loop PIDs")
            print("Thrust: {0} PWM ({1} hover + {2} delta), roll: {3:.4f}°,
pitch: {4:.4f}°, yaw: {5:.4f}°"
                  .format(thrust_CMD, HOVER_THRUST, thrust_CMD-
HOVER_THRUST, roll_CMD, pitch_CMD, yaw_CMD))
            print("Angle errors -> roll {0:.4f}°, pitch: {1:.4f}°, yaw:
{2:.4f}°"
                  .format(roll_PID.error, pitch_PID.error, yaw_PID.error))
            print("")

            print("Inner loop PIDs")
            print("Rate commands -> roll rate: {0:.3f}°, pitch rate:
{1:.3f}°, yaw rate: {2:.4f}"
                  .format(roll_rate_CMD, pitch_rate_CMD, yaw_rate_CMD))
            print("Angle rate errors -> roll rate: {0:.4f}°/s, pitch rate:
{1:.4f}°/s, yaw rate: {2:.4f}°/s"
                  .format(math.degrees(roll_rate_PID.error/1000), math.de-
grees(pitch_rate_PID.error/1000), math.degrees(yaw_rate_PID.error/1000)))
            print("")

            print("Motor 1 at ", motor_PWM[0], " PWM (thrust: ", thrust_PWM,
" roll: ", -roll_PWM, " pitch: ", pitch_PWM, " yaw: ", yaw_PWM, ")")
```

```
            print("Motor 2 at ", motor_PWM[1], " PWM (thrust: ", thrust_PWM,
" roll: ", -roll_PWM, " pitch: ", -pitch_PWM, " yaw: ", -yaw_PWM, ")")
            print("Motor 3 at ", motor_PWM[2], " PWM (thrust: ", thrust_PWM,
" roll: ", roll_PWM, " pitch: ", -pitch_PWM, " yaw: ", yaw_PWM, ")")
            print("Motor 4 at ", motor_PWM[3], " PWM (thrust: ", thrust_PWM,
" roll: ", roll_PWM, " pitch: ", pitch_PWM, " yaw: ", -yaw_PWM, ")")
            print("")

            print("Loop time: {0:.6f} s (loop start {1:.6f} + outer PIDs:
{2:.6f} + inner PIDs: {3:.6f} + motors: {4:.6f} + loop end: {5:.6f})"
                  .format(cycle_time, pre_PID_time, outer_PID_time, in-
ner_PID_time, update_motors_time, post_PID_time))
            print("")


            #input ("") #DEBUG: remove "#" to look at the cycle step by step
            #break      #DEBUG: remove "#" to check first loop only

            post_PID_time = time.process_time() - time_buoy

            ##### ELAPSE STEP TIME #####
            cycle_time = time.process_time() - cycle_start
            time.sleep(T-cycle_time)


        time.sleep(2*T) #to make sure there is no overlap with the previous
thrust command
        stop_motors(scf)
        time.sleep(0.1) #give ample time to the CF to actually stop the mo-
tors
        set_motor_control_mode(scf, AUTOMATIC_CONTROL)
        time.sleep(0.1)
        exit()


if __name__ == "__main__":

    read_thread = threading.Thread(target=read_user_input)
    main_thread = threading.Thread(target=main_program)

    main_thread.start()
    read_thread.start()

    main_thread.join()
    read_thread.join()
```

Bibliografia

[i] https://forum.bitcraze.io/
[ii]https://www.bitcraze.io/documentation/tutorials/getting-started-with-crazyflie-2-x/
[iii]https://www.bitcraze.io/support/f-a-q/#how-durable-is-the-crazyflie-2-x-
[iv]https://www.bitcraze.io/documentation/hardware/crazyflie_2_1/crazyflie_2_1-datasheet.pdf
[v]https://www.bitcraze.io/documentation/hardware/crazyradio_pa/crazyradio_pa-datasheet.pdf
[vi]https://www.bitcraze.io/documentation/repository/crazyflie-firmware/master/functional-areas/sensor-to-control/state_estimators/#extended-kalman-filter
[vii]https://www.bitcraze.io/documentation/system/platform/cf2-coordinate-system/
[viii]Julian Förster con supervisione di Michael Hamer e Prof. Dr. Raffaello d'Andrea, *System Identification of the Crazyflie 2.0 Nano Quadrocopter - Capitolo 4*, Istitute for Dynamic Systems and Control, Swiss Federal Institute of Technology (ETH) Zurich, Agosto 2015
[ix]James Diebel, *Representing Attitude: Euler Angles, Unit Quaternions, and Rotation Vectors*, Stanford University, 20 Ottobre 2006
[x]Julian Förster con supervisione di Michael Hamer e Prof. Dr. Raffaello d'Andrea, *System Identification of the Crazyflie 2.0 Nano Quadrocopter*, Istitute for Dynamic Systems and Control, Swiss Federal Institute of Technology (ETH) Zurich, Agosto 2015
[xi]Gonzalo A. Garcia, A Ram Kim, Ethan Jackson, Shawn S. Keshmiri, and Daksh Shukla, *Modeling and Flight Control of a Commercial Nano Quadrotor*, 2017 International Conference on Unmanned Aircraft Systems (ICUAS), Miami Florida, 13-16 Giugno 2017
[xii]https://www.bitcraze.io/documentation/repository/crazyflie-firmware/master/functional-areas/pwm-to-thrust/
[xiii]https://www.bitcraze.io/documentation/repository/crazyflie-firmware/master/functional-areas/sensor-to-control/controllers/#cascaded-pid-controller
[xiv]https://github.com/pms67/PID
[xv]https://github.com/bitcraze/crazyflie-lib-python
[xvi]https://www.bitcraze.io/documentation/repository/crazyflie-lib-python/master/api/cflib/
[xvii]https://www.bitcraze.io/documentation/repository/crazyflie-lib-python/master/user-guides/python_api/#variables-and-logging
[xviii]https://www.bitcraze.io/documentation/repository/crazyflie-lib-python/master/user-guides/python_api/#uniform-resource-identifier-uri
[xix]https://www.bitcraze.io/documentation/repository/crazyflie-lib-python/master/api/cflib/utils/uri_helper/
[xx]https://www.bitcraze.io/documentation/repository/crazyflie-clients-python/master/userguides/userguide_client/#firmware-configuration
[xxi]https://www.bitcraze.io/documentation/repository/crazyflie-firmware/master/api/logs/
[xxii]https://www.bitcraze.io/documentation/repository/crazyflie-firmware/master/api/params/
[xxiii]https://www.bitcraze.io/documentation/repository/crazyflie-lib-python/master/user-guides/python_api/#variables-and-logging