



UNIVERSITÀ
degli STUDI
di CATANIA

CORSO DI LAUREA MAGISTRALE (LM-18) IN DATA SCIENCE

PROGETTO DI SISTEMI ROBOTICI

Simulazione grafica 2D
di un multirrotore
con Python e PHIDIAS

Professore

Chiar.mo Prof. Corrado Santoro

Studente

dott. Pierpaolo Gumina

Gennaio 26, 2023

Sommario

Nell'elaborato presentato, si è deciso di volgere lo studio alla descrizione e analisi del movimento di piattaforme multirotore tramite modellazione e simulazione grafica. In particolare si è analizzata la simulazione grafica (bidimensionale) di un multirotore, implementando il controllo in velocità e posizione dell'asse di roll e della z .

Per dimostrare la corretta taratura¹ dei controllori si sono successivamente prodotti i grafici di velocità e posizione.

Il progetto ha inoltre come scopo la realizzazione di un ambiente bidimensionale popolato da ostacoli fissi e da oggetti da catturare.

Infine, in PHIDIAS, si è implementato l'algoritmo del cammino minimo e la strategia di movimento del drone. In particolare la strategia si compone delle due seguenti procedure:

- *generate()* - che genera 10 blocchi da posizionare in 10 posizioni sul terreno stabilite a priori generando casualmente il colore²;
- *scan and pick()* - che consente al multirotore di effettuare la scansione, blocco per blocco, ed il prelevamento del blocco solo se esso è di colore rosso o verde; il blocco catturato va poi depositato nel contenitore.

¹Per la taratura del multirotore si sono utilizzati i seguenti parametri:

- Massa del multirotore, $1.7kg$
- Coefficiente di attrito viscoso, $7 \cdot 10^{-5}$
- Forza di spinta massima motori, $60N$
- Inclinazione massima asse di roll, 20°
- Velocità angolare massima asse di roll, $80\text{gradi}/s$
- Velocità massima asse Z, $1.5m/s$
- Accelerazione/decelerazione, a piacere

²I colori possibili assumibili sono: rosso, verde e blu

Elenco delle figure

1.1	Le oscillazioni di un velivolo: yaw, pitch e roll	6
1.2	L'inclinazione delle pale di un drone	7
1.3	Tipologie di droni	7
1.4	Movimento di Yaw, rotazione attorno all'asse Z	9
1.5	Movimento di Roll, rotazione attorno all'asse X	9
1.6	Movimento di Pitch, rotazione attorno all'asse Y	10
1.7	Mixer	10
1.8	Matrice M	11
1.9	Schema del Rate Control	11
1.10	Schema dell'Attitude Control	12
2.1	Rappresentazione della posa di un multirotore.	13
2.2	Dinamica Rotazionale di un multirotore.	14
2.3	Grafico della Dinamica Rotazionale di un multirotore.	14
2.4	Dinamica Traslazionale di un multirotore.	15
2.5	Grafico che mostra l'implementazione del controllo dell'angolo θ e della velocità angolare ω	16
2.6	Grafico che mostra l'implementazione del controllo della v_x	17
2.7	Grafico che mostra l'implementazione della traslazione verticale.	18
2.8	Grafico che mostra l'implementazione del controllore Proporzionale.	21
2.9	Grafico che mostra l'implementazione del controllore Integrale.	22
2.10	Grafico che mostra l'implementazione del controllore Proporzionale-Integrale.	23
2.11	Grafico che mostra l'implementazione del controllore Proporzionale-Integrale-Derivativo.	24
2.12	Grafico che mostra le tre fasi caratterizzanti di un Profilo di Velocità	25
2.13	Grafico che mostra la fase di decelerazione	26
2.14	Grafici che mostrano il fenomeno dell'overlap	27
2.15	Grafico che mostra il controllo tramite Virtual Robot	27
2.16	Grafici in modalità PID : a destra la velocità v_x , in centro la posizione x , a destra l'angolo θ	28

2.17	Grafici in modalità Profilo di Velocità : a destra la velocità v_x , in centro la posizione x , a destra l'angolo θ	28
2.18	Grafici in modalità Virtual Robot : a destra la velocità v_x , in centro la posizione x , a destra l'angolo θ	28
2.19	Grafico di z a sinistra e di v_z a destra	28
3.1	Immagine che mostra l'ambiente grafico.	31
3.2	Diagramma che mostra l'intera implementazione in PHIDIAS.	34

Indice

Elenco delle figure	2
1 Cinematica, dinamica e controllo di un multirrotore	5
1.1 Sistema di Riferimento	5
1.1.1 Vincoli	6
1.1.2 Movimenti	8
1.1.3 Rate e Attitude Control	11
2 Simulazione e controllo di un multirrotore	13
2.1 Dinamica Rotazionale	13
2.2 Dinamica Traslazionale	14
2.3 Implementazione Python	15
2.3.1 quadrotor_model.py	15
2.3.2 angle_control.py - controllo dell'angolo e della velocità angolare	16
2.3.3 x_control.py - traslazione orizzontale	17
2.3.4 z_control.py - traslazione verticale	18
2.3.5 autopilot.py	19
2.3.6 pose.py, block.py, tower.py	20
2.3.7 world.py	20
2.3.8 gui.py	20
2.4 Grafici di velocità e posizione	21
2.4.1 Controllo Proporzionale-Integrale-Derivativo (PID)	21
2.4.2 Profilo di Velocità	24
2.4.3 Profilo con Virtual Robot	26
3 PHIDIAS: Pianificazione del Percorso e Strategia	29
3.1 phidias_interface.py	29
3.2 strategia.py	30
4 Conclusioni	35
4.1 Possibili sviluppi futuri	35

Capitolo 1

Cinematica, dinamica e controllo di un multirrotore

Un quadricottero multirrotore è un sistema bionico, che si muove nello spazio tridimensionale; è formato da un insieme di eliche identiche in numero pari, posizionate in una forma circolare. La struttura meccanica è generalmente simmetrica, con il centro di gravità che coincide col centro geometrico.

Si tratta di un velivolo a decollo e atterraggio verticali, VTOL (acronimo inglese di Vertical Take-Off and Landing), senza bisogno di una pista. Ha quattro gradi di libertà, si può muovere nelle 3 direzioni più l'heading (orientamento).

Durante le simulazioni si usano spesso i droni e non gli elicotteri perché hanno meno problemi da un punto di vista meccanico/aerodinamico e sono semplici da realizzare in quanto ci sono solo quattro motori e quattro eliche; tutta la gestione dei movimenti è ottenuta tramite il software, modulando la velocità delle 4 eliche.

1.1 Sistema di Riferimento

Nella modellazione di un quadrirotore riveste un ruolo importante il concetto di centro geometrico che corrisponde con il centro di massa del drone, caratterizzato dalle coordinate (x, y, z) ; inoltre gli angoli di Eulero determinano il comportamento del drone e sono:

- L'angolo di rollio (roll) è l'inclinazione rispetto all'asse orizzontale.
- L'angolo di beccheggio (pitch) o di pitch è la rotazione rispetto all'asse Y .
- L'angolo di imbardata (yaw) è l'inclinazione del muso rispetto ad un ipotetico nord, solitamente si usa il nord magnetico misurato con una bussola.

La posa di un multirrotore è rappresentata da: $X, Y, Z, \Phi, \theta, \Psi$, dove X, Y, Z sono le coordinate geografiche.

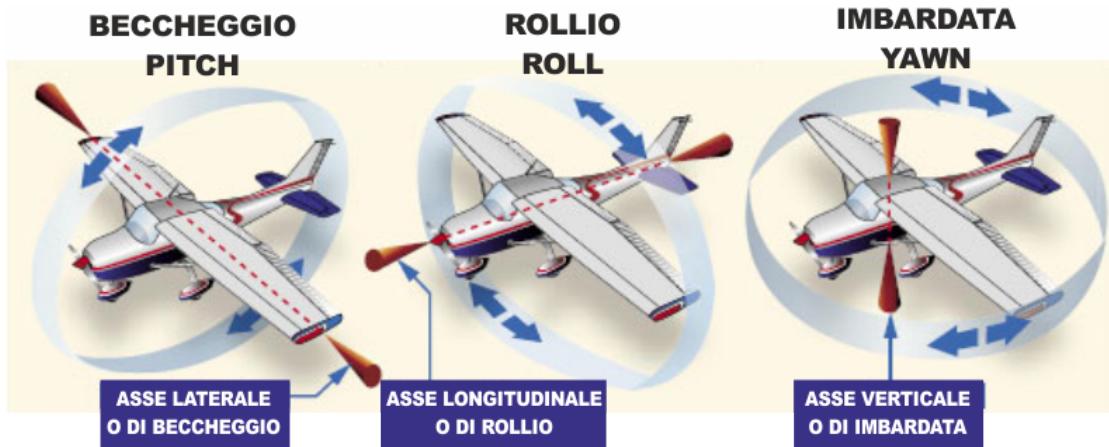


Figura 1.1: Le oscillazioni di un velivolo: yaw, pitch e roll

1.1.1 Vincoli

Durante la progettazione di un multirotore bisogna prendere in considerazione i seguenti vincoli:

- I motori e le eliche devono essere uguali;
- Le eliche devono essere maggiori o uguali a quattro;
- Le eliche devono essere collocate in posizione circolare;
- Le eliche devono ruotare in direzioni opposte in coppia (per la terza legge di Newton, il principio di azione e reazione): se ruotassero insieme, tutto il corpo ruoterebbe in verso opposto;
- Le pale sono un leggermente inclinate e le inclinazioni delle pale sono opposte a coppie, in maniera tale che il movimento possa permettere portanza verso l'alto;



Figura 1.2: L'inclinazione delle pale di un drone

- Il numero e la posizione dei motori definisce il modello airframe.

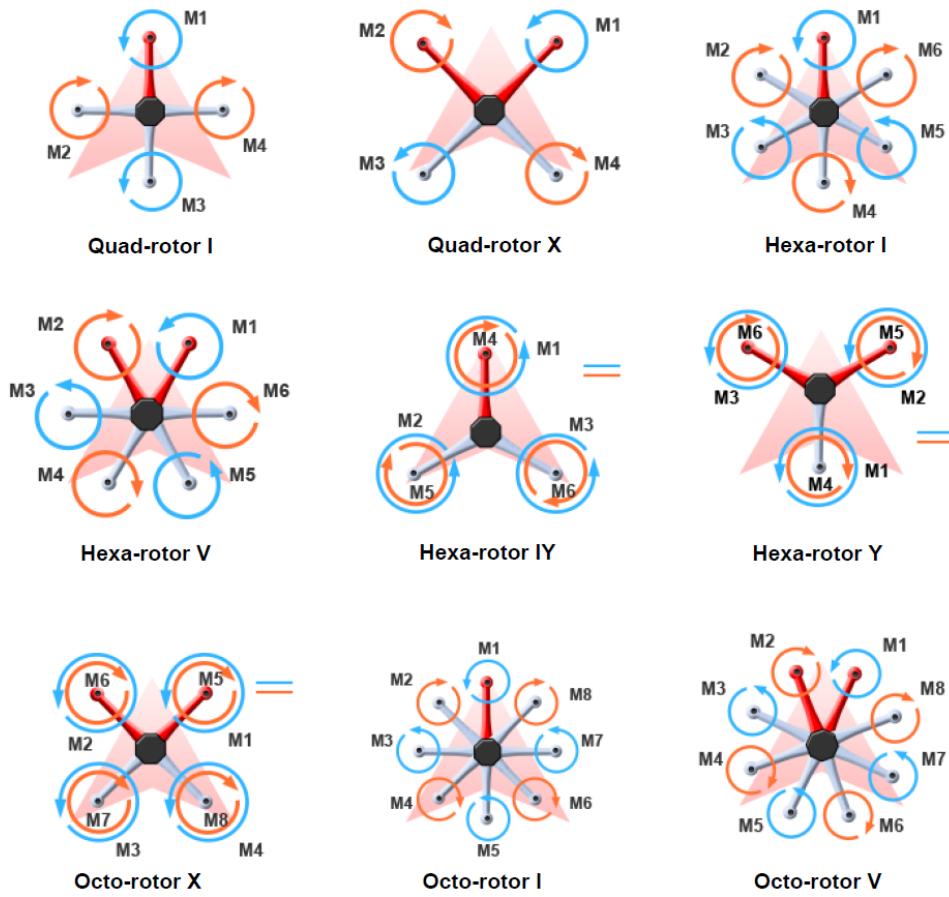


Figura 1.3: Tipologie di droni

1.1.2 Movimenti

Il movimento è ottenuto modulando le velocità delle eliche. In generale, come comandi per i sistemi bionici (e navali) si hanno fondamentalmente i seguenti quattro elementi:

- Potenza dei motori, “thrust”;
- Gli angoli di roll e pitch modificati con il “control joke”;
- L’angolo di yaw modificato con la pedaliera.

I range sono i seguenti:

$$\begin{aligned} thrust - cmd &\in [0, TH_{max}] \\ roll - cmd &\in [-R_{max}, R_{max}] \\ pitch - cmd &\in [-P_{max}, P_{max}] \\ yaw - cmd &\in [-Y_{max}, Y_{max}] \end{aligned}$$

Movimento di Hovering e traslazione rispetto all’asse Z

Il comando di *thrust* viene impostato uguale per tutti e quattro i motori tramite una distribuzione simmetrica del comando di potenza dei motori. All’aumentare del valore di *thrust*, per tutte le eliche, si ottiene una spinta verso l’alto del drone. Con il termine *hovering* si intende il veicolo fermo a mezz’aria e si ottiene sempre modulando i motori simultaneamente, riuscendo a compensare così il peso del drone.

$$\begin{aligned} \omega_1 &= thrust - cmd \\ \omega_2 &= thrust - cmd \\ \omega_3 &= thrust - cmd \\ \omega_4 &= thrust - cmd \end{aligned}$$

Movimento di Yaw, rotazione attorno all’asse Z

La velocità di rotazione a coppia dev’essere identica, si sottrae e somma un valore proporzionale allo *yaw - cmd*; quindi si combina con il comando di spinta:

$$\begin{aligned} \omega_1 &= thrust - cmd - yaw - cmd \\ \omega_2 &= thrust - cmd + yaw - cmd \\ \omega_3 &= thrust - cmd - yaw - cmd \\ \omega_4 &= thrust - cmd + yaw - cmd \end{aligned}$$

Il segno dipende dal verso di rotazione scelto.

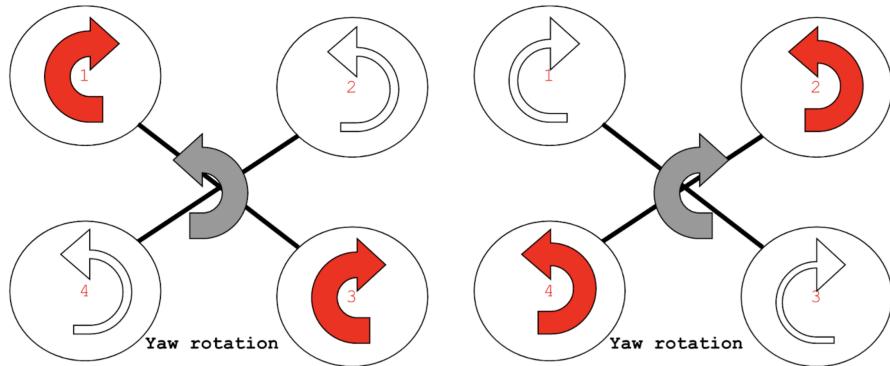


Figura 1.4: Movimento di Yaw, rotazione attorno all'asse Z

Movimento di Roll, rotazione attorno all'asse X

Le traslazioni si ottengono ruotando lungo l'asse X e l'asse Y. La rotazione lungo X si ottiene modulando le eliche 1/4 e 2/3 in modo differente.

$$\omega_1 = \text{thrust} - \text{cmd} - \text{yaw} - \text{cmd} + \text{roll} - \text{cmd}$$

$$\omega_2 = \text{thrust} - \text{cmd} + \text{yaw} - \text{cmd} - \text{roll} - \text{cmd}$$

$$\omega_3 = \text{thrust} - \text{cmd} - \text{yaw} - \text{cmd} - \text{roll} - \text{cmd}$$

$$\omega_4 = \text{thrust} - \text{cmd} + \text{yaw} - \text{cmd} + \text{roll} - \text{cmd}$$

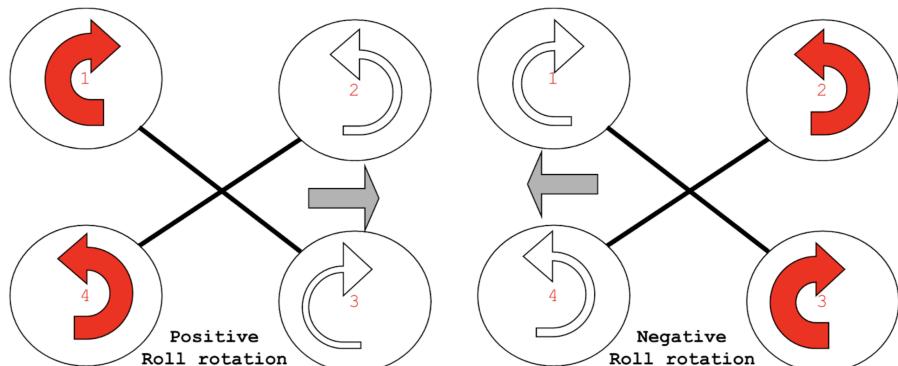


Figura 1.5: Movimento di Roll, rotazione attorno all'asse X

Se due eliche allineate spingono di più delle altre il sistema ruoterà. Se si ruota il velivolo, la spinta su Z non è più perfettamente verticale, quindi c'è un movimento lungo Y. L'angolo di *roll* e la coordinata Y sono legati, non è possibile modificare la posizione lungo Y senza agire sull'angolo di *roll*.

Movimento di Pitch, rotazione attorno all'asse Y

Per la rotazione lungo l'asse di *pitch*, si considerano le coppie di eliche 1/2 e 3/4, davanti e dietro. Se quelle davanti girano più velocemente, il muso si inclina

verso il basso, al contrario per quelle dietro.

$$\omega_1 = \text{thrust} - \text{cmd} - \text{yaw} - \text{cmd} + \text{roll} - \text{cmd} + \text{pitch} - \text{cmd}$$

$$\omega_2 = \text{thrust} - \text{cmd} + \text{yaw} - \text{cmd} - \text{roll} - \text{cmd} + \text{pitch} - \text{cmd}$$

$$\omega_3 = \text{thrust} - \text{cmd} - \text{yaw} - \text{cmd} - \text{roll} - \text{cmd} - \text{pitch} - \text{cmd}$$

$$\omega_4 = \text{thrust} - \text{cmd} + \text{yaw} - \text{cmd} + \text{roll} - \text{cmd} - \text{pitch} - \text{cmd}$$

La rotazione lungo l'asse di pitch permette una traslazione lungo X.

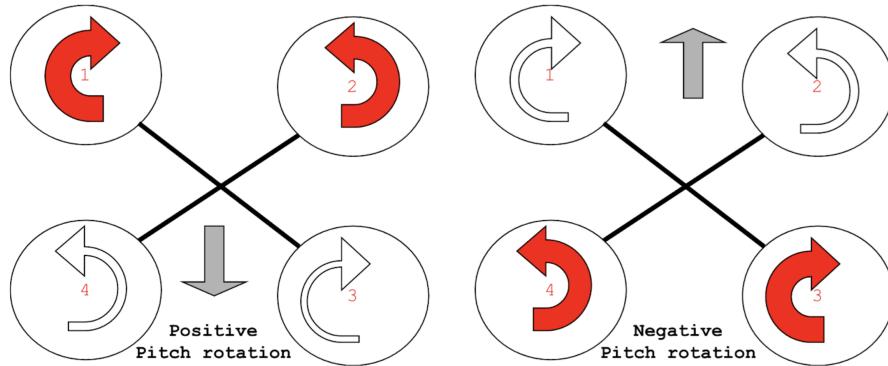


Figura 1.6: Movimento di Pitch, rotazione attorno all'asse Y

Movimento: il mixer

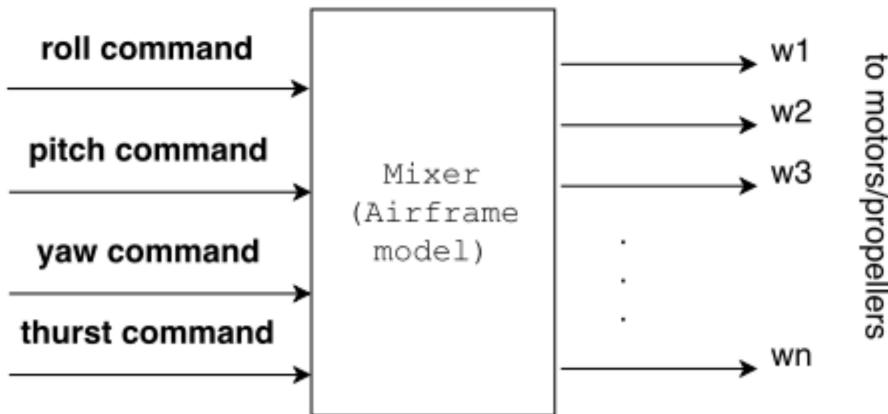


Figura 1.7: Mixer

Il *mixer* è il componente software che traduce i comandi di assetto in comandi motore. Dipende dal modello airframe e implementa fondamentalmente una matrice M tale che:

$$\begin{bmatrix} \omega_1 \\ \omega_2 \\ \dots \\ \omega_n \end{bmatrix} = M \begin{bmatrix} roll_cmd \\ pitch_cmd \\ yaw_cmd \\ thrust_cmd \end{bmatrix}$$

Figura 1.8: Matrice M

1.1.3 Rate e Attitude Control

Il *mixer* produce valori PWM e non ha il controllo delle forze reali delle eliche. Per garantire la stabilità, è necessario impiegare sensori adeguati che rilevano l'assetto del multirotore. Il controllo della stabilità si ottiene mediante due loop di controllo:

- Rate Control - controlla le velocità angolari $\bar{\Phi}, \bar{\theta}, \bar{\Psi}$ attraverso un giroscopio triassiale (3-axis gyro);
- Attitude Control - controlla gli angoli di Eulero Φ, θ, Ψ attraverso sensori come 6-DOF o 9-DOF IMU.

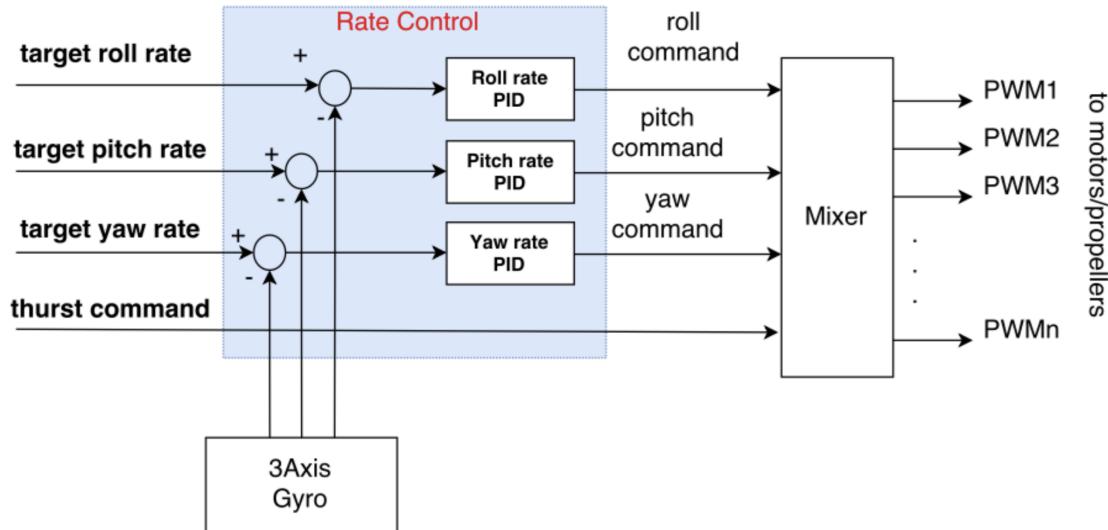


Figura 1.9: Schema del Rate Control

Il modulo Rate Control esegue un controllo PID sulle velocità angolari $\bar{\Phi}, \bar{\theta}, \bar{\Psi}$ sulla base di:

- Target Rates, dati in ingresso;
- Current Rates, dati provenienti dal giroscopio.

Il rate control assicura che le velocità angolari reali siano quelle desiderate, ma non fornisce garanzie sull'assetto, cioè che la posa dell'airframe data una specifica $\bar{\Phi}$, $\bar{\theta}$, $\bar{\Psi}$. Inoltre, partendo da una posa orizzontale $\Phi = 0$, $\theta = 0$, se un *glitch*¹ sposta improvvisamente l'angolo di beccheggio a $\theta = 10^\circ$, il sistema è stabile dal punto di vista del controllo della velocità (non ci sono rotazioni), ma (probabilmente) l'assetto non è quello desiderato.

È necessario un controllore d'assetto (Attitude Controller), che comandi i controllori di velocità e assicuri il rispetto di una certa posa, dal punto di vista degli angoli di Eulero. A questo scopo, l'assetto corrente viene stimato integrando i dati provenienti da giroscopi, accelerometri e magnetometri.

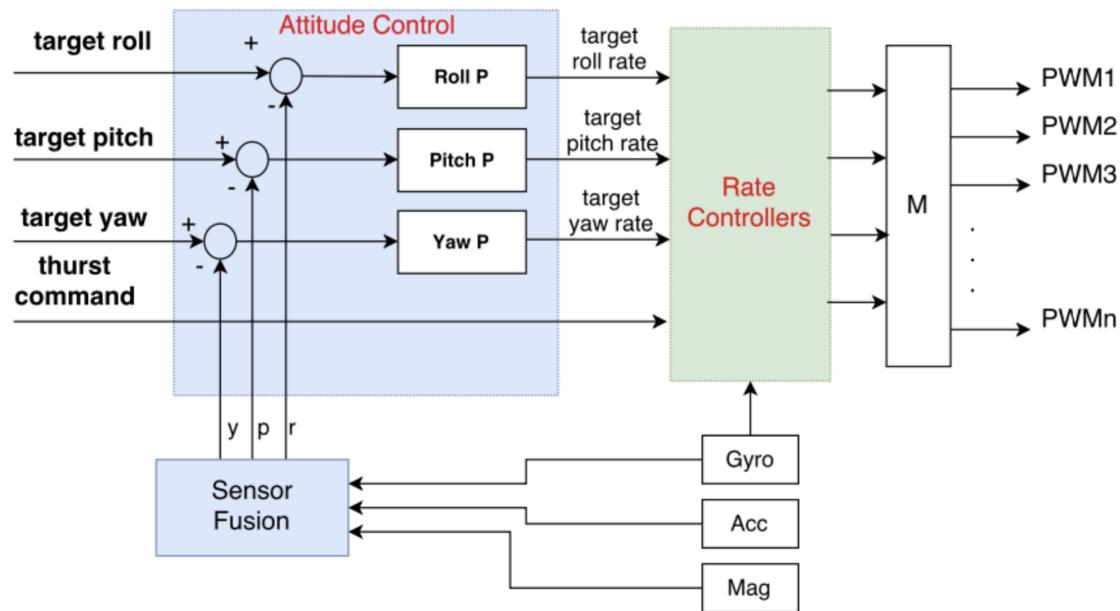


Figura 1.10: Schema dell'Attitude Control

Gli Attitude Controllers sono (di solito) semplici regolatori proporzionali (P). Le uscite sono saturate fino a una velocità angolare massima, determinata sperimentalmente.

L'Attitude Control è implementato come task periodico (come il controllo della velocità), con un periodo uguale (o superiore) a quello del controllo della velocità.

¹In elettronica, termine onomatopeico che indica genericamente i disturbi di breve durata che si manifestano in un impulso teletrasmesso, deformandone la forma d'onda.

Capitolo 2

Simulazione e controllo di un multirrotore

Per poter capire e rappresentare nel corso della trattazione gli aspetti dinamici di un quadrirotore si utilizza un modello bidimensionale semplificato. Si rappresenta la **posa** tramite una terna X, Z, θ , in cui Z è l'altitudine e θ è l'inclinazione rispetto all'orizzonte.

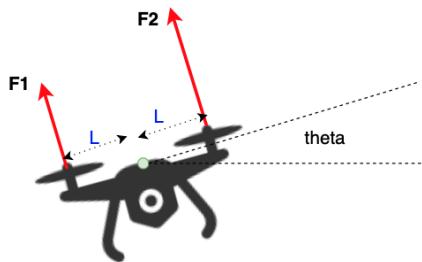


Figura 2.1: Rappresentazione della posa di un multirrotore.

Per la traslazione si utilizza la legge di Newton, somma delle forze uguale massa per accelerazione, per le rotazioni si utilizzano i momenti di inerzia e le coppie di forze.

2.1 Dinamica Rotazionale

Ogni elica (supponiamo solo 2) produce una forza perpendicolare al suo piano di rotazione. Se le forze non sono uguali, il risultato è una rotazione lungo il centro di massa; la velocità di rotazione dipende dalla differenza delle forze. Per modellare il moto utilizziamo la seconda legge di Newton: la somma delle coppie è uguale al momento di inerzia del corpo per l'accelerazione angolare. Dati:

- M , la massa del multirrotore;

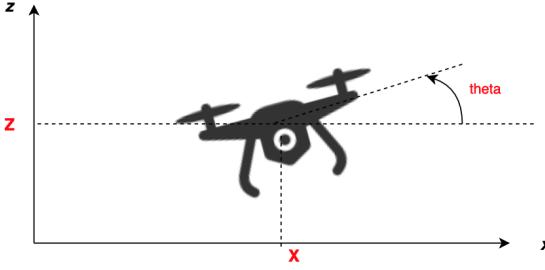


Figura 2.2: Dinamica Rotazionale di un multirrotore.

- $I = \frac{1}{12}M(2L)^2$, il momento di inerzia del multirrotore;
- $-bL\bar{\theta}$, coppia di attrito.

Si ha:

$$F_2L - F_1L - bL\dot{\theta} = I\ddot{\theta}$$

Includendo la velocità angolare $\omega = \dot{\theta}$, si ottiene:

$$\begin{cases} \ddot{\theta} = \omega \\ \dot{\omega} = -\frac{bL}{I} + \frac{L}{I}(F_2 - F_1) \end{cases}$$

Discretizzando con il rapporto incrementale si ottiene:

$$\begin{cases} \theta(k+1) = \theta(k) + \Delta T \omega(k) \\ \omega(k+1) = \omega(k) - \frac{bL\Delta T}{I} \omega(k) + \Delta T \frac{L}{I} (F_2 - F_1) \end{cases} \quad (2.1)$$

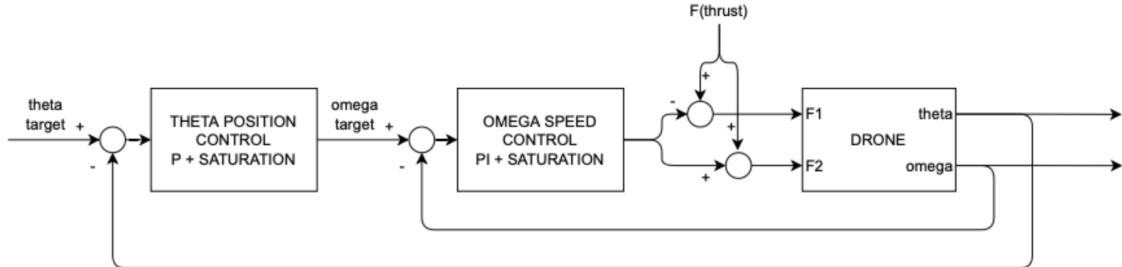


Figura 2.3: Grafico della Dinamica Rotazionale di un multirrotore.

2.2 Dinamica Traslazionale

Per semplificare la trattazione, si suppone che il multirrotore sia un singolo punto. Le due forze di spinta si compongono, come se fossero applicate entrambe al centro.

$$\begin{aligned} F_z &= (F_1 + F_2) \cdot \cos\theta \\ F_x &= -(F_1 + F_2) \cdot \sin\theta \end{aligned}$$

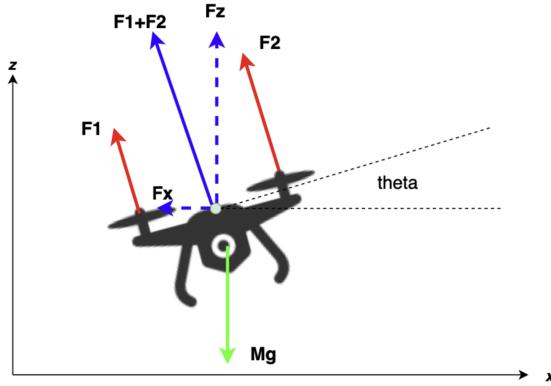


Figura 2.4: Dinamica Traslazionale di un multirotore.

La forza risulta di verso opposto al verso dell'angolo.

$$(F_1 + F_2) \sin(-\theta) - bv_x = M \dot{v}_x$$

Si ha la forza lungo x e la parte di attrito viscoso proporzionale alla velocità. Si ottiene il seguente sistema dinamico:

$$\begin{cases} \dot{v}_x = v_x \\ \dot{v}_x = -\frac{b}{M}v_x + \frac{F_1+F_2}{M} \sin(-\theta) \end{cases} \quad (2.2)$$

Discretizzando:

$$\begin{cases} x(k+1) = x(k) + \Delta T v_x(k) \\ v_x(k+1) = (1 - \Delta T \frac{b}{M}) v_x(k) + \Delta T \frac{F_1+F_2}{M} \sin(-\theta) \end{cases} \quad (2.3)$$

Per la z , bisogna considerare anche la forza peso che si va ad opporre alla spinta.

$$\dot{v}_z = -\frac{b}{M}v_z + \frac{F_1+F_2}{M} \cos\theta - g$$

Discretizzando si ottiene:

$$\begin{cases} z(k+1) = z(k) + \Delta T v_z(k) \\ v_z(k+1) = (1 - \Delta T \frac{b}{M}) v_z(k) + \Delta T \frac{F_1+F_2}{M} \cos\theta - \Delta T g \end{cases} \quad (2.4)$$

L'input saranno le due forze, gli output le posizioni e velocità.

2.3 Implementazione Python

Nel seguente paragrafo vengono descritte le implementazioni Python del progetto.

2.3.1 quadrotor_model.py

Il costruttore prende come argomenti la massa (`_m`) e la lunghezza dei bracci (`_L`). Dopo di che si inizializzano le seguenti costanti:

- Dalla letteratura si ottiene il momento di inerzia, I ;
- Le variabili di stato, ω e θ ;
- Coefficiente di attrito dinamico, b .

Nel metodo *evaluate* si introducono le equazioni che aggiornano la traslazione lungo l'asse z con z e v_z , la traslazione lungo l'asse x con x e v_x , la rotazione dinamica con θ e ω .

Si introduce un vincolo relativo alla quota zero, il robot quindi non oltrepasserà tale quota prevenendo che la z diventi negativa. Infine, il metodo *paint* ha lo scopo di disegnare il drone nell'interfaccia grafica, considerando l'eventuale situazione di blocco catturato.

2.3.2 angle_control.py - controllo dell'angolo e della velocità angolare

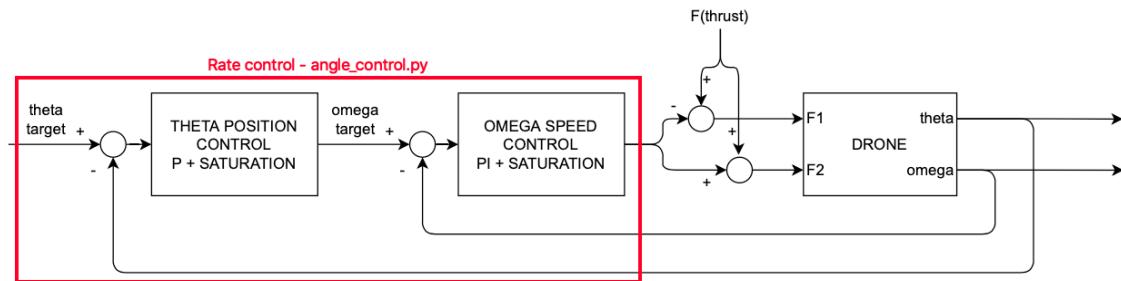


Figura 2.5: Grafico che mostra l'implementazione del controllo dell'angolo θ e della velocità angolare ω .

La classe angle control contiene un riferimento del multirrotore (_multirotor), contiene anche i coefficienti relativi ai blocchi di controllo:

- il coefficiente proporzionale del controllo in posizione (_kp_theta)
- il coefficiente proporzionale del controllo in velocità (_kp_omega)
- il coefficiente proporzionale integrale del controllo in velocità (_ki_omega)
- il coefficiente derivativo del controllo in velocità (_kd_omega)
- la saturazione della velocità angolare, ovvero il valore massimo consentito della velocità angolare (_omega_sat)
- la saturazione della forza dei motori delle eliche (_force_sat)

Il metodo **evaluate**, prende come argomento `theta_target` e il tempo di campionamento (`delta_t`). Si implementa quindi il controllo in posizione, si calcola l'errore del controllore in posizione e lo si passa al controllo in velocità, il quale restituirà la spinta da applicare (`delta_f`). L'output dell'`angle_control` (`delta_f`) andrà miscelato con la spinta dei motori e applicato infine al drone.

2.3.3 x_control.py - traslazione orizzontale

La v_x è dipendente in qualche modo dall'angolo θ . Se il drone è orientato di un certo angolo θ rispetto all'orizzonte si avrà la comparsa di una v_x , quest'ultima opposta rispetto all'inclinazione. Per poter controllare la v_x è necessario quindi andare a modulare l'inclinazione. Anche in questo caso si tende a considerare un controllo in posizione e velocità, considerando v_x e x in cascata.

Data una certa posizione `x_target` viene data in input a un controllore proporzionale con saturazione che restituisce come output una `v_x_target`, la quale viene manipolata da un controllore proporzionale integrale derivativo (PID) con saturazione che restituisce l'angolo al quale il drone dovrà inclinarsi per ottenere quella velocità target.

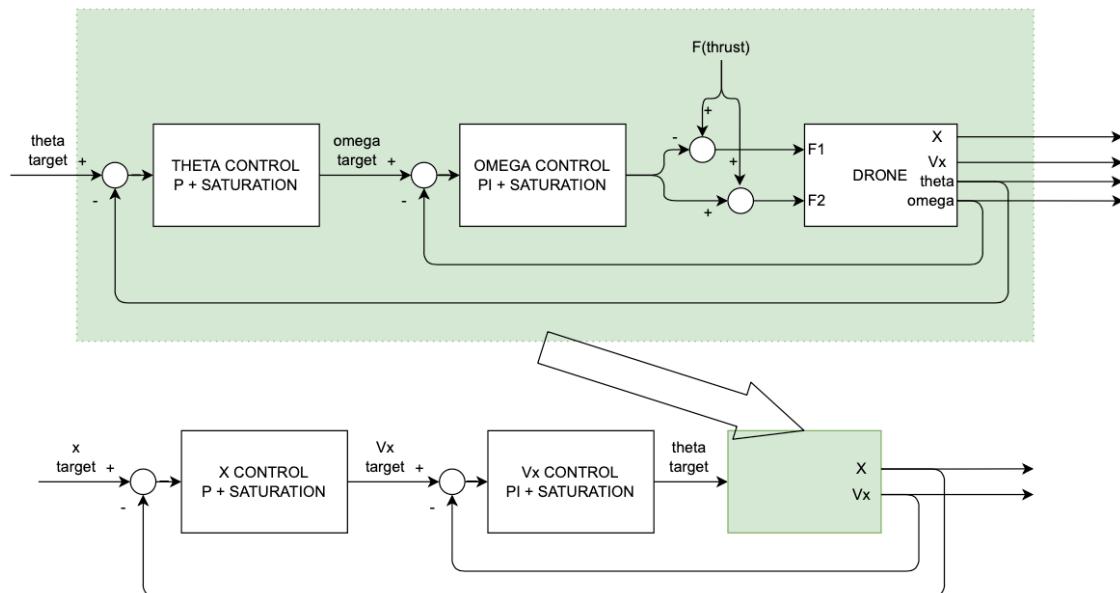


Figura 2.6: Grafico che mostra l'implementazione del controllo della v_x

La classe `x_controller` prende come input un riferimento del multirrotore (`_multirotor`), contiene anche i coefficienti relativi ai blocchi di controllo:

- il coefficiente di proporzionalità per la posizione (`_kp_x`)
- il coefficiente di proporzionalità per la velocità (`_kp_vx`)

- il coefficiente di integrazione per la velocità ($_ki_v_x$)
- il coefficiente di derivativo per la velocità ($_kd_v_x$)
- la saturazione per la v_x ($_v_x_sat$)
- la saturazione per la θ ($_theta_sat$)

Il metodo **evaluate** prende come input una x_target e il tempo di campionamento (δ_t). Si eseguono in due controllori che restituiscono un θ_{target} che verrà successivamente passato all' $\angle_control$ al fine di ottenere una data rotazione e conseguentemente una data velocità e posizione.

2.3.4 z_control.py - traslazione verticale

La z agisce sulla spinta dei motori:

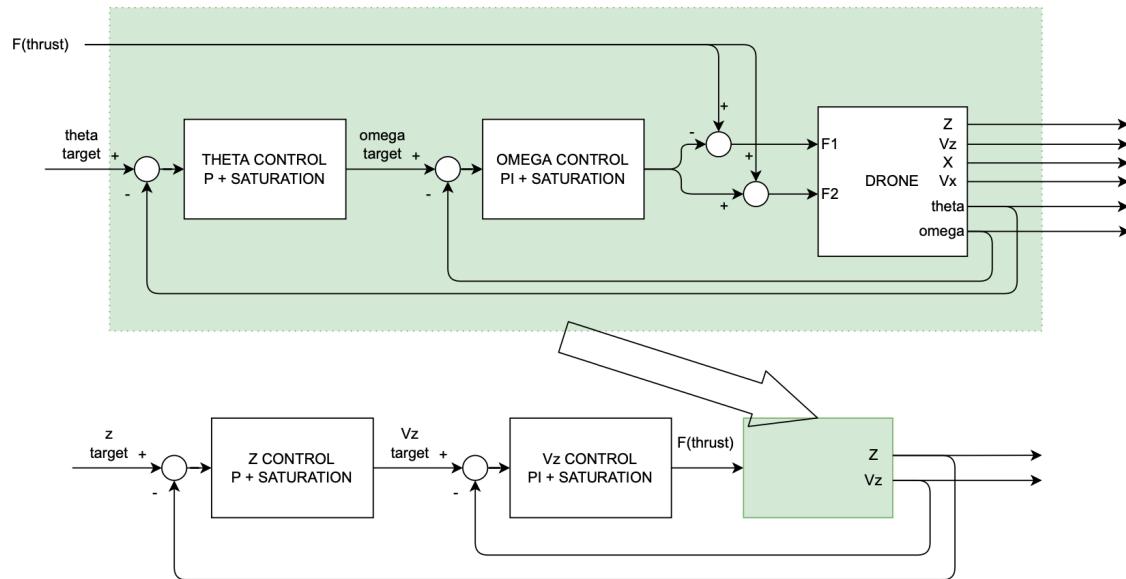


Figura 2.7: Grafico che mostra l'implementazione della traslazione verticale.

Anche in questo caso si ha uno schema di controllo di posizione e velocità, l'output della coppia di controllori in cascata produce la spinta da applicare ai motori, la quale viene poi "sbilanciata" in base al valore di θ_{target} che si vuole ottenere.

La classe `z_control` prende come input un riferimento del multirrotore (`_multirotor`), contiene anche i coefficienti relativi ai blocchi di controllo:

- il coefficiente di proporzionalità per la posizione ($_kp_z$)

- il coefficiente di proporzionalità per la velocità (`_kp_vz`)
- il coefficiente di integrazione per la velocità (`_ki_vz`)
- la saturazione per la v_z (`_vz_sat`)
- la saturazione della potenza dei motori (`_power_sat`)

Nel metodo **evaluate** accetta come argomento una quota `z_target` e il tempo di campionamento (`delta_t`). Si eseguono i due controllori che restituiscono la potenza (power) da applicare per ottenere la quota `theta_target` desiderata.

2.3.5 autopilot.py

Tutte le classi precedenti vengono richiamate all'interno della classe **Autopilot**. I dati da usare sono i seguenti:

- Massa del multirrotore, $1.7kg$
- Coefficiente di attrito viscoso, $7 \cdot 10^{-5}$
- Forza di spinta massima motori, $60N$
- Inclinazione massima asse di roll, 20°
- Velocità angolare massima asse di roll, $80\text{gradi}/s$
- Velocità massima asse Z , $1.5m/s$
- Accelerazione e decelerazione a piacere

Nel metodo **run** viene eseguito il controllo e la simulazione del multirrotore. In particolare si ottiene la potenza da applicare ai motori sulla base della quota da raggiungere, si ottiene l'inclinazione da computare lungo l'asse x (qui è presente un meno $(-)$ poiché la v_x risulta avere segno opposto rispetto all'inclinazione). Dopodiché, ottenuto il `theta_target`, si esegue l'`angle_control` ottenendo lo sbilanciamento richiesto per quell'angolo. Si applica quindi, lo sbilanciamento utilizzando il valore power meno lo sbilanciamento per l'elica 1 e power più lo sbilanciamento per l'elica 2.

Grazie alla presenza di una soglia impostata a < 0.1 (10 cm), è possibile contrassegnare la posizione target come raggiunta quando il drone è abbastanza vicino. Infine con il metodo `set_target` è possibile definire le coordinate x, y del punto target.

2.3.6 pose.py, block.py, tower.py

La classe **Pose**, prende in input come parametro il puntatore all'oggetto che identifica l'interfaccia grafica (`_ui`), e ha il compito di posizionare i vari oggetti all'interno dell'ambiente. Con il metodo `set_pose` è possibile assegnare una nuova posa all'oggetto. Il metodo `get_pose` restituisce la posizione attuale dell'oggetto all'interno dell'ambiente.

All'interno di `pose.py` è presente inoltre il metodo `to_pixel` che permette di convertire le unità di misura fisiche del sistema internazionale in coordinate pixel di schermo. Inoltre, la posa degli oggetti all'interno della scena sono espressi in termini di x, y, α ; dove x, y rappresentano le coordinate e α l'angolo di orientamento.

La classe **block.py** ha lo scopo di creare e posizionare i vari blocchi all'interno della grafica. Questa classe dispone inoltre del metodo `get_color`, necessario per l'identificazione del colore del blocco.

La classe **tower.py**, infine, ha lo scopo di posizionare la torre all'interno della quale verranno collocati i vari blocchi una volta presi.

2.3.7 world.py

La classe **world.py** raggruppa le classi precedentemente descritte per formare l'intera interfaccia di simulazione. Essa dispone dei seguenti metodi:

- `paintObstacles` - posiziona gli ostacoli all'interno della scena;
- `new_block` - aggiunge un nuovo blocco all'interno della scena. Prende come parametri di input il colore del blocco e lo slot su cui posizionarlo.
- `pop_block` - rimuove un blocco dalla scena.
- `sense_color` - restituisce il colore del blocco più vicino al drone.

2.3.8 gui.py

Si tratta di una classe grafica che, attraverso le librerie qt, permette di visualizzare i vari oggetti. In particolare con $\delta_t = 1 \cdot e^{-3}$ si fa' partire un timer che per ogni millisecondo viene eseguito il metodo `go` che non fa altro che eseguire il metodo `run` di autopilot e aggiornare l'aspetto grafico.

La classe `gui` dispone dei seguenti metodi:

- Il metodo `go_to_nodo` prende come input il nodo di arrivo e lo aggiunge come target alla classe autopilot. Questo metodo viene richiamato dal file `strategia.py`, ovvero dall'interprete PHIDIAS (vedi paragrafo successivo) per spostare il drone in un nodo specifico;

- Il metodo `notify_target_got` ha lo scopo di informare l'interprete PHIDIAS del raggiungimento del nodo target.
- Il metodo `set_held_block` ha lo scopo di prelevare il blocco una volta che il drone è posizionato di sopra.
- Il metodo `release_block_to_tower` ha lo scopo di rilasciare il blocco nella torre di rilascio.
- Il metodo `generate_blocks` ha lo scopo di posizionare 10 blocchi nei rispettivi slot, randomizzando la scelta dei colori.
- Il metodo `sense_color`, una volta invocato, restituisce all'interprete PHIDIAS il colore del blocco più vicino al drone.

2.4 Grafici di velocità e posizione

A questo punto lo studio volge l'attenzione alla taratura del multirotore. In particolare si sono prodotti grafici per tre diversi modelli:

- controllo Proporzionale Integrale Derivativo;
- profilo di velocità;
- profilo virtual robot.

2.4.1 Controllo Proporzionale-Integrale-Derivativo (PID)

Il controllo proporzionale-integrale-derivativo (in breve PID) si compone delle seguenti tre componenti:

- Parte proporzionale (P) - ha il compito di decrementare la spinta man mano ci si avvicina alla posizione target, cercando di essere in accordo con la dinamica del sistema che non riesce a fermarsi istantaneamente.

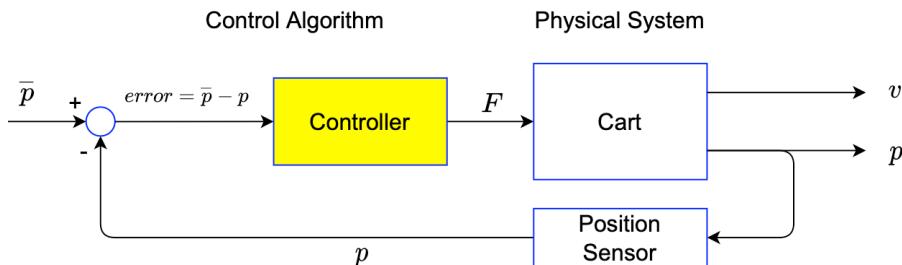


Figura 2.8: Grafico che mostra l'implementazione del controllore Proporzionale.

Il controllore risponde alla seguente equazione:

$$f(k) = k_p(\text{target} - p(k))$$

Bisogna scegliere il valore di K_p adeguatamente: più è alta questa costante, più il controllore degenera in un controllore di tipo on/off. La scelta dipende da linee guida basate su test pratici. Imporre una forza proporzionale all'errore funziona, ma bisogna modellarlo bene perché in certi casi il sistema impiega troppo tempo a raggiungere la posizione e in altri casi, se la costante è alta, potrebbe oscillare molto.

Questo nel grafico si traduce nella comparsa di una **sovraelongazione**, cioè si supera la posizione target per poi tornarci. Il sistema sta spingendo eccessivamente ai pressi della posizione target. Diminuendo la costante K_p , diminuiscono le sovraelongazioni, ma allo stesso tempo il sistema impiega più tempo a raggiungere il target. Bisogna sempre trovare il giusto compromesso.

- Parte integrale (I) - in questo contesto il problema del controllo è quello della velocità; si vuole che il robot raggiunga una velocità ben precisa. All'interno del controllore si installa un sensore che legge la velocità attuale e viene sottratta ad una velocità target precedentemente impostata, ottenendo la spinta $f(k)$ necessaria.

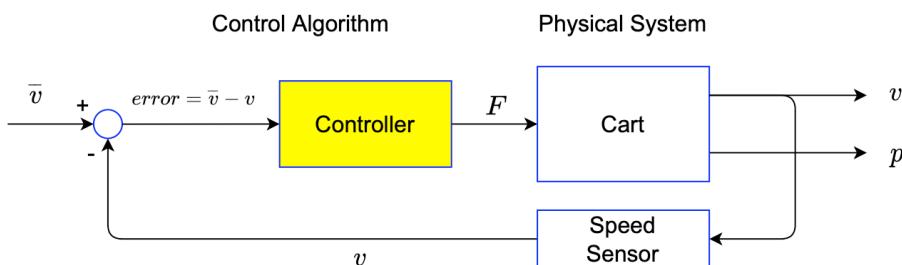


Figura 2.9: Grafico che mostra l'implementazione del controllore Integrale.

Nel controllo della posizione, la formula del controllore proporzionale asserisce che quando il target è raggiunto, il sistema dev'essere fermo e quindi $f(k) = 0$, mentre se è maggiore di zero si ottiene una spinta positiva, e se è minore di zero una spinta negativa.

Discorso differente nel controllo della velocità, quando la velocità target desiderata è raggiunta il sistema deve rimanere in moto a velocità costante, il semplice controllore proporzionale restituirebbe 0 come errore: il sistema non spingerebbe, la velocità diminuirebbe e il target non sarebbe nuovamente raggiunto.

È necessario un accumulatore di errore, per cui la spinta non deve variare ad errore 0.

$$f(k) = f(k - 1) + Qe(k)$$

Dove il coefficiente Q fa variare la spinta in modo proporzionale all'errore. Il controllore è un accumulatore di errore quindi è un integratore; la spinta al valore t è uguale alla spinta al valore $t - 1$.

Dal punto di vista analitico si ottiene:

$$f(t) = K_I \int_0^t e(\tau) d\tau$$

Discretizzando si ottiene:

$$f(k) = K_I \sum_{j=0}^k e(j) \Delta t$$

Combinando il controllore proporzionale e il controllore integrale si ottiene la seguente formula di aggiornamento:

$$f(t) = K_p e(t) + K_I \int_0^t e(\tau) d\tau$$

Discretizzando si ottiene:

$$f(k) = K_p e(k) + K_I \sum_{j=0}^k e(j) \Delta t$$

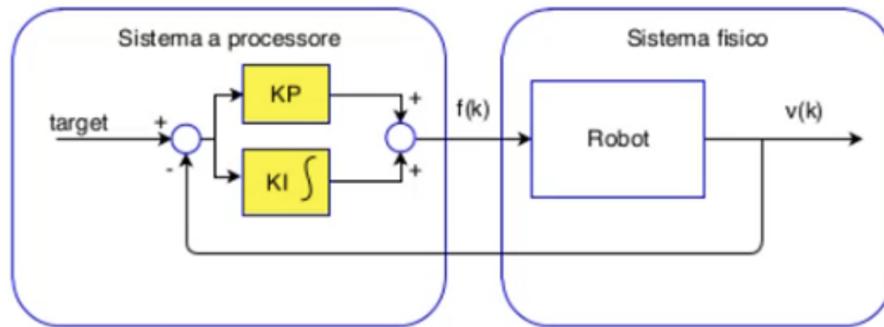


Figura 2.10: Grafico che mostra l'implementazione del controllore Proporzionale-Integrale.

dove k_p e k_I sono le **costanti di controllo** che modellano l'errore per avere una forza desiderata. C'è quindi un fattore accumulativo proporzionale sommato ad uno "integrale", implementato mediante una variabile che accumula l'errore moltiplicato per l'intervallo di campionamento. Il fattore proporzionale è detto a breve termine, quello integrale a lungo termine, dato che accu-mula tutti gli errori degli istanti precedenti. Nella fase iniziale agisce di più la componente proporzionale e la velocità cresce rapidamente, successivamente agisce la componente integrale e la crescita diminuisce.

- Parte derivativa (D) - l'azione derivativa è spesso definita "anticipata" in quanto è una sorta di stima del futuro. L'azione derivativa è utile per i sistemi lenti, solitamente quelli meccanici con elevata inerzia (soprattutto quando sono sistemi oscillanti): per questi sistemi servono azioni forti, di "spingere"

molto il controllore aumentando le costanti. Un esempio è un pendolo molto pesante che dev'essere frenato in modo sostenuto prima di arrivare alla posizione target. Limitarsi ad aumentare le singole costanti P e I , in questi casi, non produce l'effetto desiderato; se da una parte aumentare K_I migliora la dinamica del sistema, dall'altra introduce oscillazioni smorzate indesiderate. Bisogna, quindi, valutare se si sta deviando troppo dalla traiettoria target e agire di conseguenza; si aggiunge una parte basata sulla derivata dell'errore rispetto al tempo.

$$K_p e + k_I \int e dt + k_d \frac{de}{dt}$$

La derivata può andare in tre direzioni: non variare, aumentare o diminuire. La tipica situazione è che l'errore diminuisce esponenzialmente, però potrebbe anche oscillare e ci sono parti con derivata positiva e parti con derivata negativa. Se l'errore diminuisce, il sistema si trova in una situazione buona di raggiungimento del target ed è possibile diminuire l'azione di controllo sottraendo un termine negativo. Se l'errore aumenta, si è in presenza di una situazione poco desiderabile e l'azione di controllo dev'essere incrementata, bisogna contrastare l'allontanamento dal target.

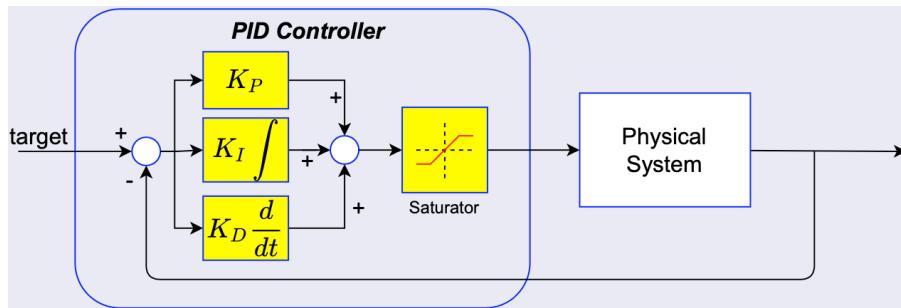


Figura 2.11: Grafico che mostra l'implementazione del controllore Proporzionale-Integrale-Derivativo.

Nei multirrotori la parte derivativa risulta fondamentale quando il multirrotore è grande. Le caratteristiche di inerzia del multirrotore sono legate alla lunghezza dei bracci: più sono corti, più il multirrotore è scattante e meno tende ad oscillare, al contrario con bracci lunghi tende ad oscillare maggiormente.

2.4.2 Profilo di Velocità

Un profilo di velocità è caratterizzato da tre fasi distinte:

1. fase di accelerazione, caratterizzata da una rampa di accelerazione;
2. fase a velocità costante;

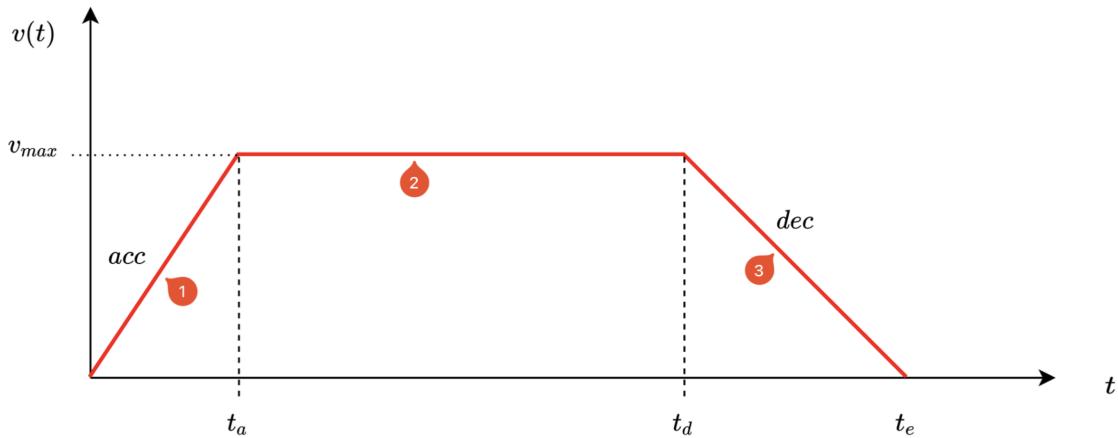


Figura 2.12: Grafico che mostra le tre fasi caratterizzanti di un Profilo di Velocità

- 3. fase di decelerazione, caratterizzata da una rampa di decelerazione.

L’obiettivo è considerato raggiunto se al termine delle tre fasi la posizione target è raggiunta. I parametri da passare al controller sono:

- la posizione target;
- il parametro di accelerazione;
- il parametro di velocità massima;
- il parametro di decelerazione.

La prima e la terza fase sono moti rettilinei uniformemente accelerati, la seconda fase è un moto rettilineo uniforme.

$$(1) a = \text{cost}$$

$$(2) v(t) = v(0) + at$$

$$(3) x(t) = x(0) + v(0)t + \frac{1}{2}at^2$$

Nella fase di accelerazione si parte da una velocità nulla e la si incrementa poco a poco, per ogni intervallo di campionamento tramite la formula (2), inoltre va controllato sempre che la velocità sia inferiore a quella massima tramite un saturatore.

Fase di decelerazione

Il controllore lavora in termini di errore di posizione, quindi è necessario capire a quale distanza dal target bisogna iniziare a decelerare: bisogna quindi determinare la distanza di decelerazione.

$$D = \frac{1}{2} \frac{v_{max}^2}{dec}$$

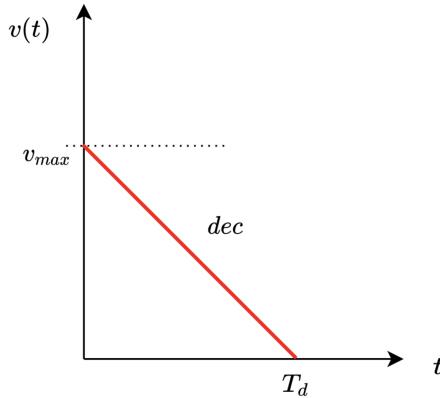


Figura 2.13: Grafico che mostra la fase di decelerazione

Il risultato è la distanza dal target da cui iniziare la fase di decelerazione. Di conseguenza se $p_{target} - p_{current} \leq D$, allora il sistema si trova nella fase di decelerazione. Successivamente bisogna determinare la velocità target da passare al controllore in velocità, per decelerare; questo tramite la seguente formula (anche in questo caso i passaggi sono omessi):

$$v = \sqrt{v_{max}^2 + 2a_3(X_D - (targetposition - currentposition))}$$

Bisogna porre particolare attenzione al segno dell'errore di posizione (dev'essere sempre positivo per avere una radice positiva): la velocità dev'essere concorde all'errore di posizione.

Non appena la distanza dal target diminuisce, la fase di accelerazione e decelerazione si accorciano al punto da sovrapporsi, questo fenomeno è chiamato **overlap**. Per ovviare a questo problema si usa una proprietà della fase 3: la fase di decelerazione impone che la velocità target sia sempre minore della velocità corrente; questa proprietà viene a mancare proprio nel punto di intersezione delle due rette. Si sfrutta, quindi, questa proprietà per interrompere la fase di accelerazione, non ancora conclusa, per iniziare quella di decelerazione.

2.4.3 Profilo con Virtual Robot

Usando come il profilo con Virtual Robot l'obiettivo è sempre quello arrivare a seguire un profilo di velocità a trapezio, con accelerazione, moto a velocità costante, e decelerazione. Utilizzando sempre una legge oraria, si immagina che il robot ideale si muova lungo la traiettoria desiderata in condizioni ideali.

La creazione di una legge oraria fornisce la posizione teorica del robot virtuale e la si confronta con la posizione del robot reale per cercare di inseguirlo. Se il robot reale è più veloce del virtuale va decelerato.

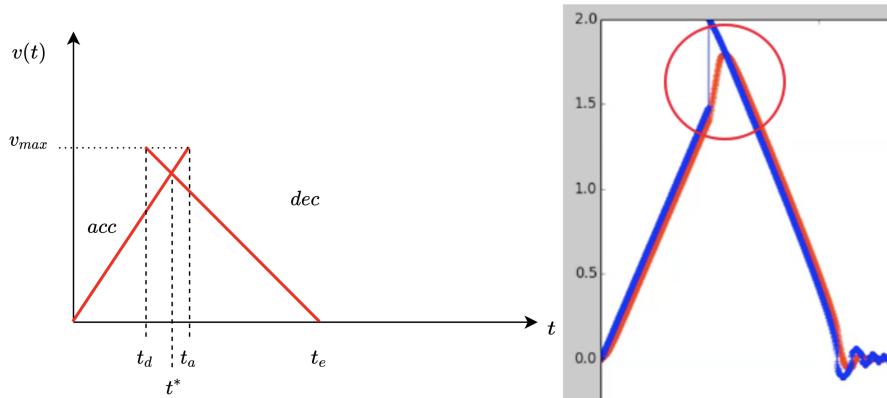


Figura 2.14: Grafici che mostrano il fenomeno dell'overlap

Il position controller questa volta è un semplice controllore proporzionale. Si calcola quindi sempre un errore di posizione e il controller calcola la velocità da avere per catturare il robot virtuale. Più è piccola la costante del controllore proporzionale, più l'inseguimento subisce ritardo, se invece la costante è troppo grande si corre il rischio di superarlo. Il moto si conclude quando il robot virtuale raggiunge la posizione finale e poi il robot reale. Si mantiene sempre un errore di posizione non nullo, il robot virtuale è sempre avanti del robot reale. La velocità target sarà diversa da zero se l'errore di posizione è diverso da zero.

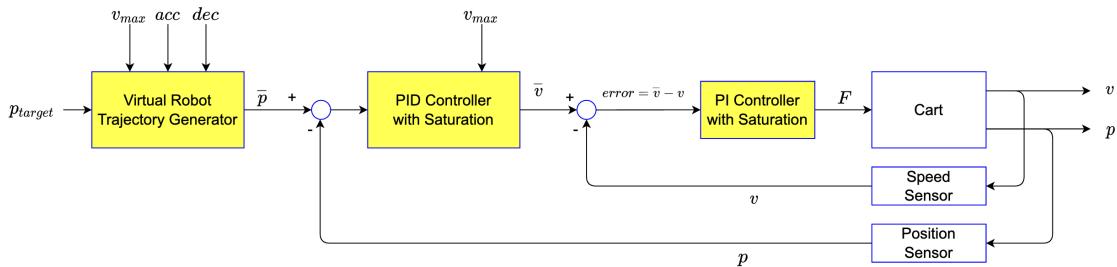


Figura 2.15: Grafico che mostra il controllo tramite Virtual Robot

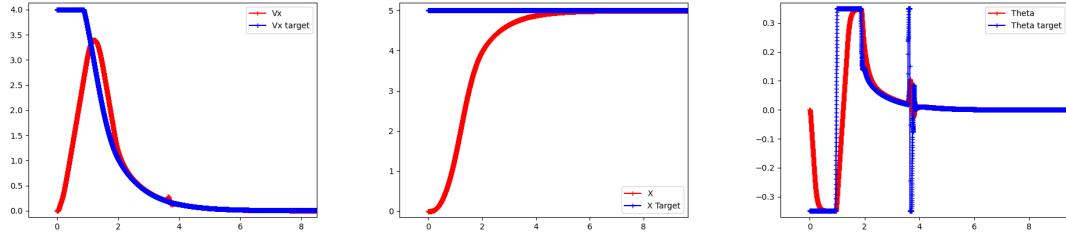


Figura 2.16: Grafici in modalità **PID**: a destra la velocità v_x , in centro la posizione x , a destra l'angolo θ .

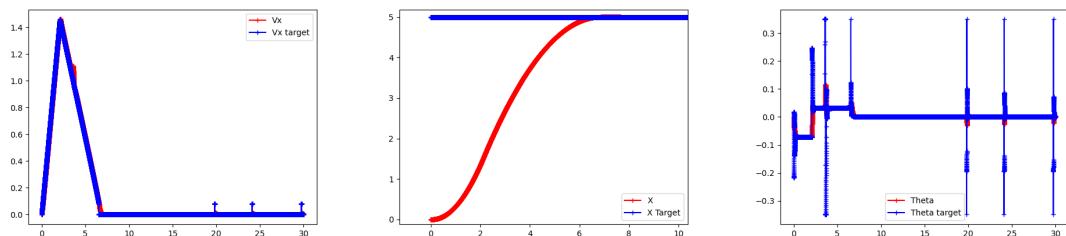


Figura 2.17: Grafici in modalità **Profilo di Velocità**: a destra la velocità v_x , in centro la posizione x , a destra l'angolo θ .

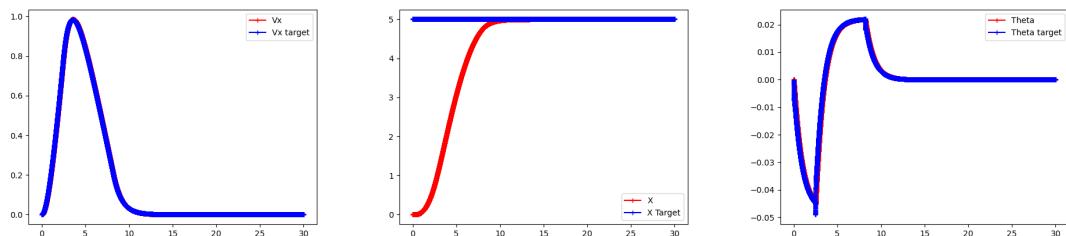


Figura 2.18: Grafici in modalità **Virtual Robot**: a destra la velocità v_x , in centro la posizione x , a destra l'angolo θ .

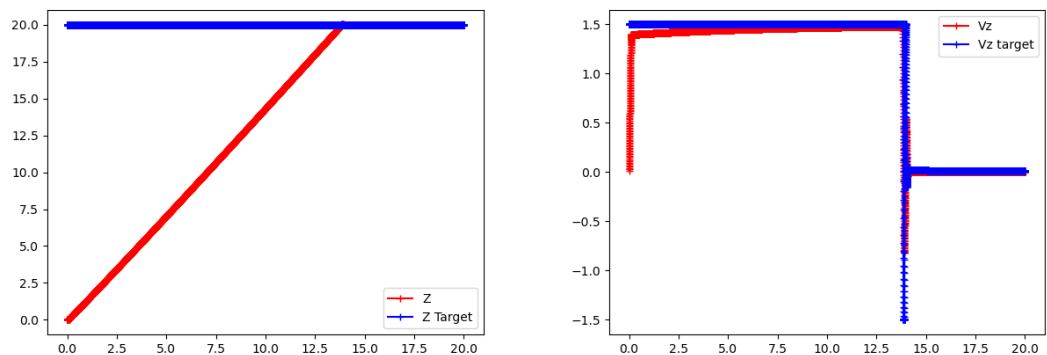


Figura 2.19: Grafico di z a sinistra e di v_z a destra

Capitolo 3

PHIDIAS: Pianificazione del Percorso e Strategia

PHIDIAS è un linguaggio di programmazione che permette di usare i costrutti PROLOG in Python; inoltre, sfrutta i paradigmi della programmazione reattiva. PHIDIAS si basa sul paradigma Belief-Desire-Intention e permette di implementare comportamenti basati su regole reattive, proattive e sulla conoscenza.

I successivi paragrafi mostrano come far comunicare l'ambiente grafico Python con quello PHIDIAS.

3.1 phidias_interface.py

Il sorgente phidias_interface ha lo scopo di far dialogare il simulatore grafico e l'interprete PHIDIAS. La classe **Messaging**, situata in fondo al file, contiene i seguenti due metodi statici:

- `parse_destination`.
- `send_belief`: ha lo scopo di inviare un belief ad un agente remoto. Dispone dei seguenti parametri:
 - `cls`: identifica la classe (`self`), viene inserito di default dal runtime PHIDIAS.
 - `destination`: è una stringa che identifica l'agente destinatario in termini di `nomeAgente@indirizzoHost:numeroPorta`. Esempio: `main@127.0.0.1:6565`.
 - `belief`: identifica il nome del belief specificato da una stringa.
 - `terms`: identifica i parametri del belief; deve essere un array python.
 - `source`: identifica il nome dell'agente che sta inviando il messaggio.

Il `send_belief` è richiamato nella classe `gui`, e permette di mandare all'agente le informazioni circa la distanza, il colore del blocco e del raggiungimento del nodo destinazione.

Il metodo `start_message_server_http` permette di avviare, nell'interfaccia grafica, un thread che implementa un server http. I parametri di ingresso sono: l'oggetto `ui` che identifica l'ambiente di simulazione e la porta `tcp` su cui avviare il server (di default 6566).

Il metodo `process_incoming_request` è una callback¹ che viene invocata nel momento in cui viene ricevuto dal server http una richiesta. In questa procedura vi è tutta l'analisi delle informazioni relative al protocollo di Messaging.

Viene interpretato, inoltre, il belief che è stato specificato, in particolare la variabile `name` rappresenta il nome del belief ricevuto e `terms` è una array che contiene i parametri del belief. In base al belief ricevuto viene richiamato uno dei seguenti metodi di `gui.py`:

- `go_to_node`
- `generate_blocks`
- `sense_color`
- `send_held_block`
- `releaseBlockToTower`

Nell'interpretazione di questi comandi è presente il metodo `ui.set_From(_From)` che ha il compito di aggiornare un attributo dell'oggetto `ui` specificando chi è l'agente che ha mandato il messaggio; serve principalmente a sapere a quale agente mandare periodicamente l'informazione di posa.

3.2 strategia.py

La prima fase consiste nell'implementare l'algoritmo del cammino minimo all'interno di un grafo. I vari nodi del grafo sono stati scelti a priori all'interno dell'interfaccia grafica rispettando i criteri di raggiungibilità e di obstacle avoidance, come mostra la figura 3.1.

¹In programmazione, un callback (o, in italiano, richiamo) è, in genere, una funzione, o un blocco di codice che viene passata come parametro ad un'altra funzione. In particolare, quando ci si riferisce alla callback richiamata da una funzione, la callback viene passata come argomento ad un parametro della funzione chiamante. In questo modo la chiamante può realizzare un compito specifico (quello svolto dalla callback) che non è, molto spesso, noto al momento della scrittura del codice.

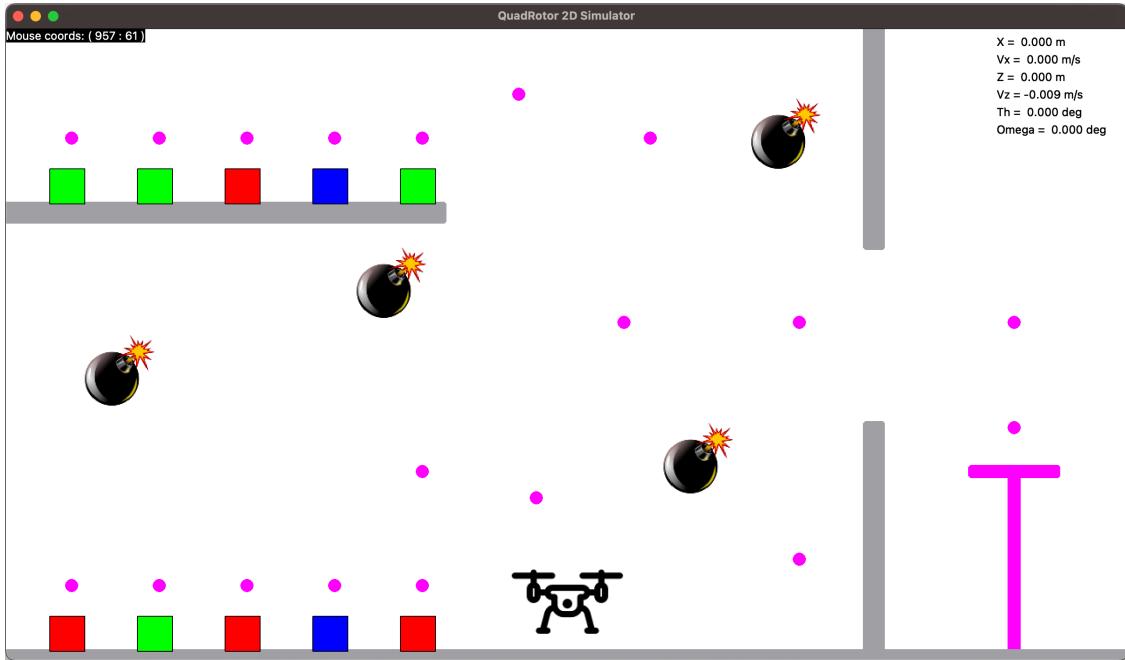


Figura 3.1: Immagine che mostra l’ambiente grafico.

Una volta posizionati i vari punti si è creato lo spanning tree ovvero tutti i possibili cammini dal nodo sorgente al nodo destinazione. Lo scopo del cammino minimo è trovare il percorso minimo da un nodo sorgente al nodo destinazione.

L’algoritmo del cammino minimo prevede di richiamare la procedura **path** ogni qualvolta si passa da un nodo al successivo; presenta quattro parametri (*P*, *Total*, *Next*, *Dest*), *P* identifica il percorso parziale intrapreso finora, *Total* identifica il costo del percorso parziale intrapreso finora, *Src* identifica il nodo corrente, *Dest* identifica il nodo destinazione. A questo punto bisogna generare altri cammini parziali e selezionare quello con il costo minimo. Questa ramificazione ricorsiva viene fatta aggiungendo una particolare annotazione PHIDIAS denominata [**’all’**].

```
path(P, Total, Src, Dest)[ ' all ' ]
    / (link(Src, Next, Cost) & nodeNotInPath(P, Next)) >> \
    [
        "P = P.copy()",
        "P.append(Src)",
        "Total = Total + Cost",
        select_min(P, Total, Next, Dest)
    ]
```

La funzione **nodeNotInPath(P,Next)** ha lo scopo di prevenire cicli all’interno del grafo. Da notare che la riga *P = P.copy()* è necessaria in quanto il percorso va copiato per tutte le ramificazioni; se fosse omesso, l’array verrebbe modificato

da tutti i percorsi mischiandosi. Con *P.append(Src)* si inserisce in coda il nodo sorgente al percorso *P*, si aggiorna il costo totale *Total*, infine la procedura **select_min** verifica se il percorso parziale in analisi risulta essere più lungo rispetto a quello precedentemente selezionato; se ciò risulta vero non si procede oltre con quel percorso, altrimenti si richiama ricorsivamente la procedura **path**.

```
select_min(P, Total, Next, Dest) /  
    (selected(CurrentMin, CurrentMinCost) &  
     gt(Total, CurrentMinCost)) >> \  
    [  
        #show_line(P, " ", Next, " ", cost " , Total, " [CUT]" )  
    ]  
  
select_min(P, Total, Next, Dest) >> \  
    [  
        path(P, Total, Next, Dest)  
    ]
```

Infine, si aggiorna il belief **selected** nel momento in cui si arriva alla fine; in quanto se l'algoritmo è arrivato alla fine vuol dire che si è superato tutti i possibili controlli di "taglio" del percorso e quindi quel percorso è un possibile candidato minimo.

```
path(P, Total, Dest, Dest) >> \  
    [  
        "P.append(Dest)" ,  
        #show_line(P, " ", Total),  
        +selected(P, Total)  
    ]
```

La procedura **generate()**, se richiamata, ha lo scopo di posizionare 10 blocchi di colore differente in 10 slot prestabiliti.

La procedura **scan_and_pick()**, consente al multirotore di effettuare la scansione, blocco per blocco, ed il prelevamento del blocco solo se esso è di colore rosso o verde; il blocco catturato va poi depositato nel contenitore. La procedura **scan_and_pick()** invoca una delle due istanze della procedura **generaESeguiPercorso** in base al possesso o meno di un blocco. In particolare, se il drone possiede un blocco allora genera e segue un percorso dalla posizione attuale al contenitore target; se invece non possiede alcun blocco deve generare e seguire il percorso verso il prossimo slot più vicino dove è situato un blocco non ancora analizzato.

Una volta che il drone è arrivato al nodo di destinazione si possono verificare due possibili scenari:

- il nodo in questione è un nodo in cui è presente un blocco; si esegue una scansione colore e si preleva il blocco se è di colore rosso o verde; se è tra i

colori obiettivi si preleva, si imposta il belief $bloccoPreso=1$ e si riesegue la procedura generaESeguiPercorso, generando un percorso verso il nodo target contenitore; se il blocco scansionato è di colore blu si ignora e si riesegue la procedura generaESeguiPercorso generando un percorso verso un altro nodo slot da scansionare.

- il nodo in questione è il nodo target contenitore; si rilascia il blocco, si imposta il belief $bloccoPreso=0$ e si riesegue la procedura generaESeguiPercorso.

Il seguente diagramma mostra l'esecuzione dell'intero programma:

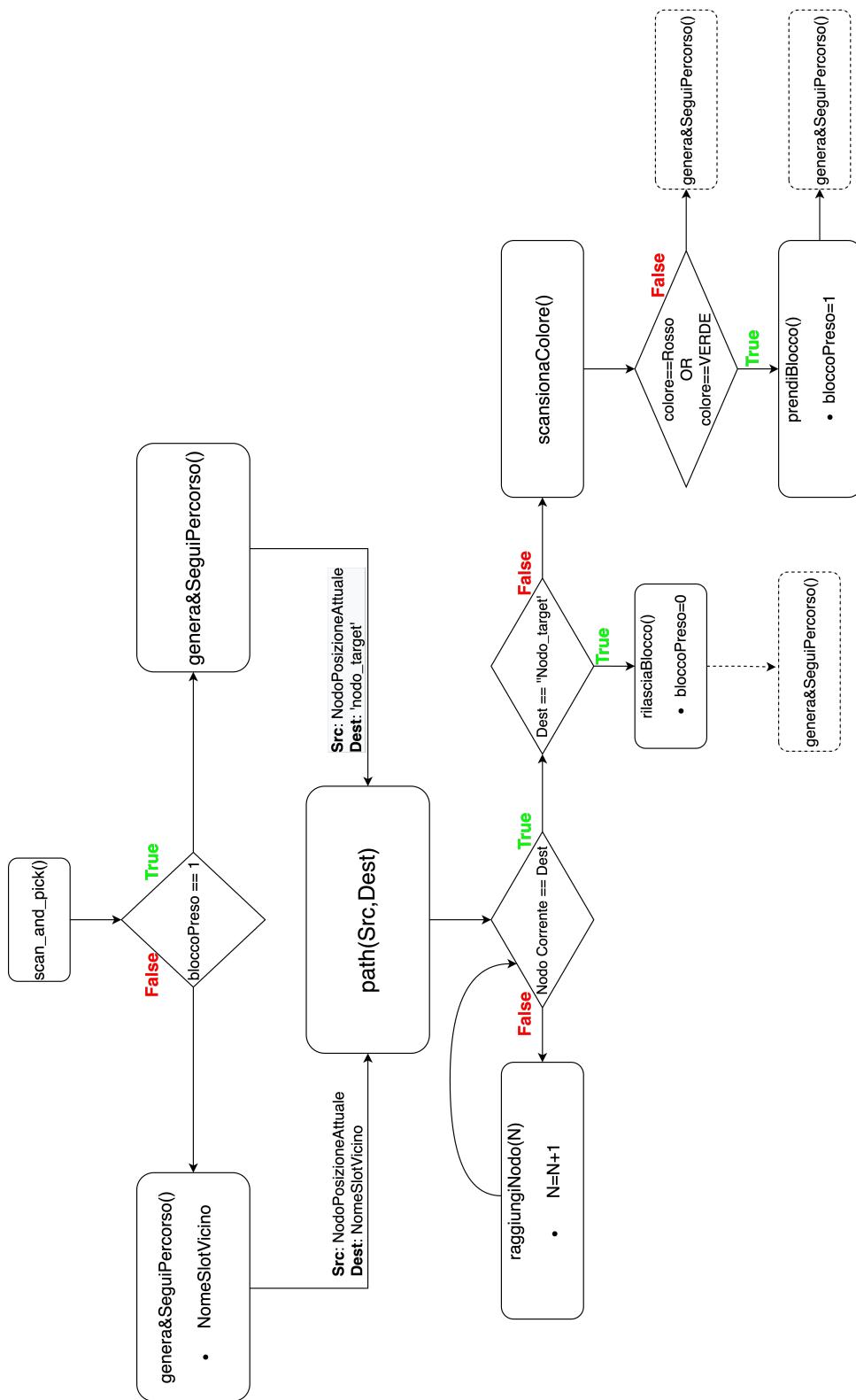


Figura 3.2: Diagramma che mostra l'intera implementazione in PHIDIAS.

Capitolo 4

Conclusioni

Al termine del lavoro di progetto si sono raggiunti tre risultati fondamentali:

- Implementare un multirrotore in Python;
- Taratura dei parametri legati ai vari controllori, producendo i relativi grafici;
- Implementare in PHIDIAS l'algoritmo del cammino minimo e la strategia di cattura.

Per questi motivi ci si può ritenere soddisfatti ed orgogliosi di aver fatto un passo avanti verso la realizzazione di un sistema multirrotore, portando al suo interno aspetti di innovazione sviluppati all'interno della facoltà, ovvero PHIDIAS.

4.1 Possibili sviluppi futuri

Scegliendo di sviluppare la simulazione grafica in un ambiente bidimensionale, non è stato possibile rappresentare tutti gli angoli di Eulero, come quello di pitch e yaw.

Una possibile implementazione futura potrebbe essere quella di ricreare la simulazione grafica del multirrotore in un ambiente tridimensionale, mediante un motore grafico 3D come **Unity** o **Unreal Engine**. Grazie all'ausilio di queste tecnologie sarebbe possibile modellare tutti gli aspetti di manipolazione di un multirrotore 3D, riuscendo a rappresentare tutti gli angoli di Eulero (roll, pitch e yaw).