Universitá Politecnica Delle Marche

Ingegneria Informatica e dell'Automazione

Manutenzione Preventiva per la robotica e l'automazione intelligente

Realizzazione osservatore diagnostico su ArduPilot SITL

Autori:
Amal Benson Thaliath
Massimo Ciaffoni
Simone Cappanera

Docente:
Prof. Alessandro Freddi

Anno Accademico 2021-2022

Indice

Capitolo 1

Installazione e configurazione Ardupilot

1.1 AirSim

AirSim è un simulatore di droni sviluppato in Unreal Engine 4, open-source e crossplatform che supporta sia simulazioni *software-in-the-loop* con PX4 o ArduPilot e simulazioni *hardware-in-the-loop*. Per il seguente progetto è stato utilizzato in modalità software-in-the-loop tramite ArduPilot per sviluppare e testare un semplice osservatore per un quadricopter.

Figura 1.1: Simulazione in AirSim

Per l'installazione seguire innanzitutto la guida ufficiale presente nel Github di AirSim:

`https://microsoft.github.io/AirSim/build_windows/`

(E' disponibile anche la versione per Linux, guida reperibile nel medesimo sito)
Al termine del processo di installazione e di building dovrebbe risultare presente la cartella del progetto Unreal `Blocks` presente in **C:/Program Files (x86)/Microsoft Visual Studio/2019/CommunityAirSim/Unreal/Environments/Blocks** dove è presente il file soluzione di Visual Studio `Blocks.sln`, oppure la cartella di progetto Unreal creata manualmente qualora non è stato possibile completare il processo di building. Lanciato il file soluzione si può avviare la simulazione una volta impostato il debugger a `Win64 Debug Game Editor`.

1.2 ArduPilot SITL

ArduPilot è un software open-source utilizzato per il controllo di aerei, droni e rover basati su sistemi ardupilot. Il software mette a disposizione anche una componente software-in-the-loop che come già accennato può collegarsi con AirSim per effettuare simulazioni di volo in modo programmato.
Per l'installazione seguire la guida ufficiale:

`https://ardupilot.org/dev/docs/sitl-native-on-windows.html`

La guida consiglia di installare ArduPilot, se si utilizza Windows, tramite il sottosistema Linux di Windows WSL1 o WSL2, tuttavia per il seguente progetto è stato seguito il metodo di installazione tramite `Cygwin`, indicato poco più sotto nella medesima guida.
ATTENZIONE: Nei passi in cui sarà necessario inserire comandi nel terminale Cygwin evitare di copiare e incollare i comandi, infatti in alcuni casi i trattini "-" vengono convertiti in caratteri simili ma non riconosciuti dal terminale. Si consiglia dunque di scrivere sempre i comandi manualmente.

1.3 Simulazione voli

Per effettuare il collegamento tra AirSim e ArduPilot ed eseguire la simulazione seguire i seguenti passi:

1. Modificare il file `settings.json` in **Documents/AirSim** con il seguente contenuto:

```
{
  "SettingsVersion": 1.2,
  "LogMessagesVisible": true,
  "SimMode": "Multirotor",
  "OriginGeopoint": {
```

```
    "Latitude": -35.363261,
    "Longitude": 149.165230,
    "Altitude": 583
  },
  "Vehicles": {
    "Copter": {
      "VehicleType": "ArduCopter",
      "UseSerial": false,
      "LocalHostIp": "127.0.0.1",
      "UdpIp": "127.0.0.1",
      "UdpPort": 9003,
      "ControlPort": 9002
    }
  }
}

```

2. Lanciare il progetto `Blocks.sln` con Visual Studio. Attendere l'avvio dell'editor Unreal Engine, all'apertura tuttavia non va premuto il tasto play dall'editor prima del terzo passo.

3. Aprire il terminale Cygwin x64, navigare nella cartella ArduPilot con `cd ArduPilot/ArduCopter` e lanciare il comando `python3.7 ../Tools/autotest/sim_vehicle.py -v ArduCopter -f airsim-copter --console --map`.
   A questo punto attendere l'avvio di MavProxy.

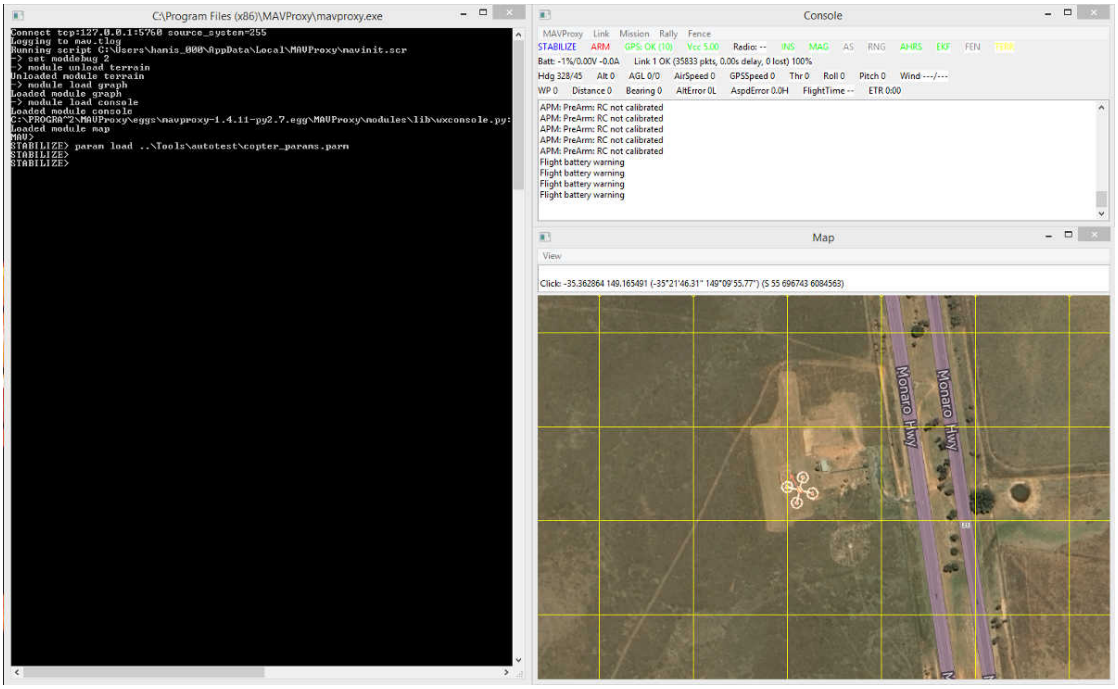


Figura 1.2: UI di MavProxy

4. Ora è possibile premere il tasto di avvio nell'editor di Unreal Engine e il collegamento sarà completato. Se la simulazione si blocca potrebbe essere il file settings.json impostato male, problemi con l'antivirus (in particolare AVG) o il firewall.

Da MavProxy è possibile creare un percorso con il click destro sulla mappa. Di seguito i comandi per accendere il drone e lanciare il percorso sono:

1. `arm throttle`
2. `mode guided`
3. `takeoff 100`
4. `long MISSION_START`
5. `mode land` (Al termine del percorso)

Per ulteriori informazioni o aggiornamenti vedere la guida ufficiale:

`https://ardupilot.org/dev/docs/sitl-with-airsim.html#using-airsim-with-ardupilot`

1.4 Modifica parametri fisici del drone

Una volta installato AirSim è neccessario modificare alcuni parametri fisici in modo che siano conformi a quelli del drone presente in laboratorio. I valori uguali o pressochè simili sono stati lasciati inalterati, mentre quelli che si discostavano significativamente sono stati modificati con i parametri del drone reale. Per poter modificare tali parametri è stato necessario recuperare quelle parti del codice che facessero riferimento alla fisica del quadricottero. Abbiamo trovato le informazioni utili in diversi file:

- *MultiRotorParams*, contenuto nella cartella **.../Blocks/Plugins/Source/AirLib/include/vehicles/multirotor**
- *RotorParams*, nella stessa cartella.

Riportiamo di seguito in dettaglio i parametri che abbiamo modificato:

- In *MultiRotorParams*:
  - alla riga 316 ad arm_lenghts il valore **0.225f** con **0.25f**;
  -alla riga 323 ad params.mass il valore **1.0f** con **1.22f**;

```
316        std::vector<real_T> arm_lengths(params.rotor_count, 0.25f);
317
318
319
320
321
322
323        params.mass = 1.22f;
```

Figura 1.3: Parametri da modificare in MultiRotorParams

- In *RotorParams*:
  - alla riga 37 a real_T C_T il valore **0.109919f** con **0.1271f**;
  - alla riga 38 a real_T C_P il valore **0.040164f** con **0.0311f**;
  - alla riga 41 real_T propeller_diameter il valore **0.2286f** con **0.2413f**.

```
37        real_T C_T = 0.1271f; // the thrust co-efficient @ 6396.667 RPM, measured by UIUC.
38        real_T C_P = 0.0311f; // the torque co-efficient at @ 6396.667 RPM, measured by UIUC.
39        real_T air_density = 1.225f; //  kg/m^3
40        real_T max_rpm = 6396.667f; // revolutions per minute
41        real_T propeller_diameter = 0.2413f; //diameter in meters, default is for DJI Phantom 2
42        real_T propeller_height = 1 / 100.0f; //height of cylindrical area when propeller rotates, 1 cm
43        real_T control_signal_filter_tc = 0.005f; //time constant for low pass filter
44
45        real_T revolutions_per_second;
46        real_T max_speed; // in radians per second
47        real_T max_speed_square;
48        real_T max_thrust = 4.179446268f; //computed from above formula for the given constants
49        real_T max_torque = 0.055562f; //computed from above formula
```

Figura 1.4: Parametri da modificare in RotorParams

1.5 Linking librerie ad Ardupilot

Per la realizzazione dell'osservatore è stato necessario utilizzare delle librerie C++ per la rappresentazione delle matrici e dei vettori e per la risoluzione di ODE, equazioni differenziali ordinarie. Le librerie messe a disposizione sono scaricabili al seguente link:

`https://www.boost.org/users/download/`

Una volta scaricate le librerie è necessario effettuare il linking con le altre librerie di Ardupilot in modo che siano utilizzabili all'interno dello stesso. Per effettuare il linking bisogna effettuare una serie di passaggi illustrati di seguito:

- Scaricato lo zip inserire la sotto-cartella boost interna contenente le librerie all'interno di **../ArduPilot/libraries**

- Inserire nel file *wscript* all'interno della cartella **../ArduPilot** la riga di codice il cui numero di riga è evidenziato in verde, visibile nello screen sottostante:

```
507     # add in generated flags
508     cfg.env.CXXFLAGS += ['-include', 'ap_config.h']
509     cfg.env.append_value('CXXFLAGS', ['-w', '-fexceptions'])
```

Figura 1.5: wscript in ../ArduPilot

- Inserire nel file *wscript* all'interno della cartella **../ArduPilot/ArduCopter** le righe dove il numero di riga è evidenziato in verde, visibili nello screen sottostante:

```
9           ap_libraries=bld.ap_common_vehicle_libraries() + [
10              'AC_AttitudeControl',
11              'AC_InputManager',
12              'AC_PrecLand',
13              'AC_Sprayer',
14              'AC_Autorotation',
15              'AC_WPNav',
16              'AP_Camera',
17              'AP_IRLock',
18              'AP_InertialNav',
19              'AP_Motors',
20              'AP_RCMapper',
21              'AP_Avoidance',
22              'AP_AdvancedFailsafe',
23              'AP_SmartRTL',
24              'AP_Beacon',
25              'AP_Arming',
26              'AP_WheelEncoder',
27              'AP_Winch',
28              'AP_Follow',
29              'AP_LTM_Telem',
30              'AP_Devo_Telem',
31              'AP_OSD',
32              'AC_AutoTune',
33              'AP_KDECAN',
34              'boost',
35          ],
36      )
37
38      bld.ap_program(
39          program_name='arducopter',
40          program_groups=['bin', 'copter'],
41          use=vehicle + '_libs',
42          defines=['FRAME_CONFIG=MULTICOPTER_FRAME'],
43          cxxflags=['-Wno-error=undef', '-w', '-Wno-error=shadow'],
44          )
45
46      bld.ap_program(
47          program_name='arducopter-heli',
48          program_groups=['bin', 'heli'],
49          use=vehicle + '_libs',
50          defines=['FRAME_CONFIG=HELI_FRAME'],
51          cxxflags=['-Wno-error=undef', '-w', '-Wno-error=shadow'],
52          )
53
54
55  def configure(ctx):
56      ctx.env.append_value('CXXFLAGS', ['-Wno-error=undef', '-w', '-Wno-error=shadow'])
```

Figura 1.6: wscript in ../ArduPilot/ArduCopter

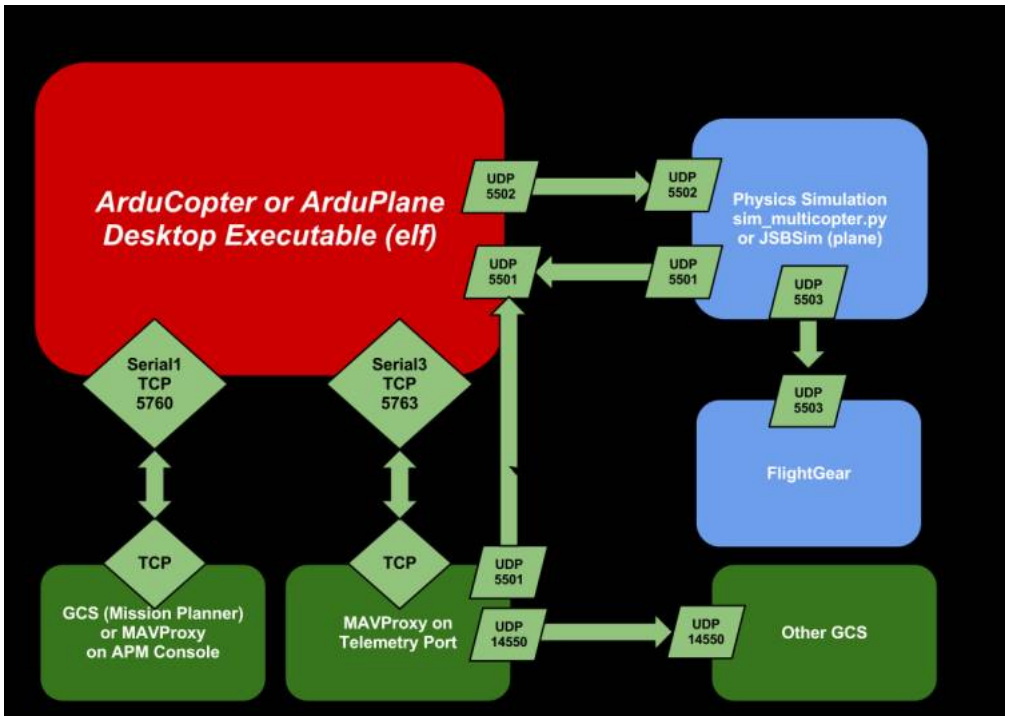
- Infine compilare le nuove librerie inserite rilanciando ArduPilot da terminale Cygwin o WSL.

Capitolo 2

Realizzazione dell'osservatore

2.1 Stima dei sensori

Per la realizzazione dell'osservatore abbiamo cercato in quale file Ardupilot ricevesse i valori calcolati nella simulazione Airsim.



Abbiamo trovato all'interno della cartella **ArduPilot/libraries/SITL** i file `SIM_Airsim.h` e la sua implementazione in cpp. `SIM_Airsim` eredita da `SIM_Aircraft` in cui sono presenti funzioni che recuperano la ground truth da AirSim e simulano le letture

dei sensori inniettando del rumore nella ground truth.
Nel codice (`SIM_Airsim.cpp`) i valori dei sensori si possono recuperare con la variabile **state**.

2.2 Calcolo dei valori di input

L'osservatore progettato richiede un vettore di input costituito dalle componenti u_f, u_p, u_q, u_r, tuttavia su ArduPilot sono disponibili solo gli input PWM. E' possibile però risalire al vettore di input u ricavandosi le formule inverse in base alla legge di controllo del drone.

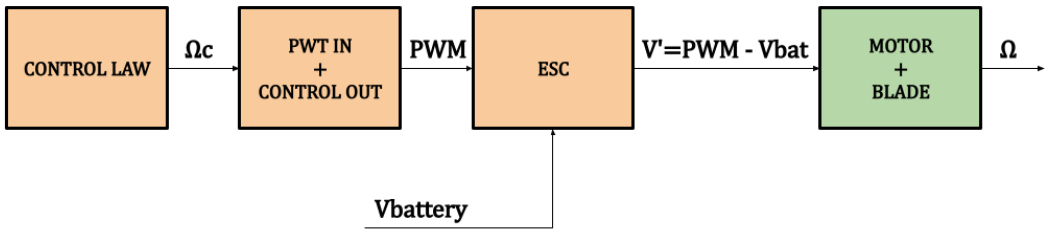


Figura 2.1: Legge di controllo dei motori

$$
\begin{aligned}
CMD_i &= \frac{PWM_i - PWM_{min}}{PMW_{max} - PWM_{min}} \quad \epsilon[0,1] \\
\Omega_i^2 &= CMD_i \cdot \Omega_{max}^2 \\
u_f &= KT \cdot \sum_{i=1}^{4} \Omega_i \\
u_p &= KT \cdot l \cdot (\Omega_4^2 - \Omega_2^2) \\
u_q &= KT \cdot l \cdot (\Omega_3^2 - \Omega_1^2) \\
u_r &= KQ \cdot (\Omega_1^2 - \Omega_2^2 + \Omega_3^2 - \Omega_4^2) \\
\Omega_{max} &= \frac{MAX\_RPM}{60} \cdot 2\pi
\end{aligned}
$$

N.B. Gli indici utilizzati precedentemente fanno riferimento al modello teorico. Nell'implementazione invece gli indici sono diversi per il fatto che vengono ordinati in maniera differente da ArduPilot nel vettore degli input PWM. I valori delle costanti sono:

- $PWM_{min} = 1000$

- $PWM_{max} = 2000$
- $KT = 1.077 \cdot 10^{-5}$
- $KQ = 9.596 \cdot 10^{-8}$
- $l = 0.25$ (lunghezza bracci del drone, parametro *arm_lengths*)
- $MAX\_RPM = 6396.667$

Questi calcoli sono implementati nella funzione `calc_input` presente nel file `Observer.h` fornito dal progetto.
ATENZIONE: Non confondere questo KT con la costante k_t (in minuscolo) utilizzata nelle matrici dell'osservatore.

2.3 Linearizzazione e definizione dell' osservatore di stato

Siano x e u rispettivamente il **vettore di stato** e il **vettore di input** di un sistema di controllo allora:

$$\Sigma : \ \dot{x} = F(x, u),$$

dove F(. , .) è una smooth function. Definiamo la coppia (x_e, u_e) un **punto di equilibrio** se F(x_e, u_e)=0. Utilizzando l'espansione di Taylor allora è possibile linearizzare Σ attorno al punto (x_e, u_e):

$$\begin{aligned}\dot{x} &= F(x_e, u_e) + \frac{\partial F(x_e, u_e)}{\partial x}(x - x_e) + \frac{\partial F(x_e, u_e)}{\partial u}(u - u_e) + h.o.t \\ &\approx \frac{\partial F(x_e, u_e)}{\partial x}(x - x_e) + \frac{\partial F(x_e, u_e)}{\partial u}(u - u_e),\end{aligned}$$

Il **modello linearizzato** allora può essere utilizzato per descrivere le relazioni tra i diversi scostamenti $\Delta x = x - x_e$ e $\Delta u = u - u_e$. Definendo le matrici A e B come

$$A = \frac{\partial F(x_e, u_e)}{\partial x} \qquad B = \frac{\partial F(x_e, u_e)}{\partial u}$$

la linearizzazione del modello Σ è dunque:

$$\Sigma_{lin} : \dot{\Delta x} = A\Delta x + B\Delta u.$$

Definiamo infine **un osservatore dello stato** come:

$$\Sigma_{obs} : \begin{aligned} \dot{\widehat{\Delta x}} &= A\widehat{\Delta x} + B\Delta u + H(\Delta x - \widehat{\Delta x}) \\ \hat{x} &= x_e + \widehat{\Delta x}, \end{aligned}$$

dove H è tale per cui A - H è una matrice Hurwitz (tutti gli autovalori di A - H sono a parte reale negativa).

2.4 Design dell'osservatore per il quadrirotore

Una volta linearizzato l'osservatore da realizzare, definiamo quali sono le componenti dell'equazione differenziale nel caso del quadricottero. Con diverse approssimazioni ed in termini di virtual input, il modello matematico che descrive quadrirotore è il seguente:

$$\begin{aligned} m\ddot{x}_F &= [\cos(\varphi)\sin(\theta)\cos(\psi) + \sin(\varphi)\sin(\psi)]u_f - k_t\dot{x}_F \\ m\ddot{y}_F &= [\cos(\varphi)\sin(\theta)\cos(\psi) - \sin(\varphi)\sin(\psi)]u_f - k_t\dot{y}_F \\ m\ddot{z}_F &= \cos(\varphi)\cos(\theta)u_f - mg - k_t\dot{z}_F \\ I_x\dot{p} &= -k_r p - qr(I_z - I_y) + u_p \\ I_y\dot{p} &= -k_r q - pr(I_x - I_z) + u_q \\ I_z\dot{r} &= -k_r r - pq(I_y - I_x) + u_r \\ \dot{\varphi} &= p + q\sin(\varphi)\tan(\theta) + r\cos(\varphi)\tan(\theta) \\ \dot{\theta} &= q\cos(\varphi) - r\sin(\varphi) \\ \dot{\psi} &= [q\sin(\varphi) + r\cos(\varphi)]/\cos(\theta), \end{aligned}$$

dove $x_F, y_F, z_F, \dot{x}_F, \dot{y}_F, \dot{z}_F, p, q, r, \varphi, \theta, \psi$ sono le variabili di stato *uf,up,uq,ur* sono i virtual input mentre le componenti rimanenti *m,g,kt,kr,Ix,Iy,Iz* sono costanti. Dato dunque il vettore di stato x e il vettore di input u, la funzione F nel caso del quadrirotore è descritta dalla seguente matrice:

$$F(x,u) = \begin{bmatrix} \dot{x}_F \\ \dot{y}_F \\ \dot{z}_F \\ (\cos(\varphi)\sin(\theta)\cos(\psi) + \sin(\varphi)\sin(\psi))(u_f/m) - (k_t/m)\dot{x}_F \\ (\cos(\varphi)\sin(\theta)\cos(\psi) - \sin(\varphi)\cos(\psi))(u_f/m) - (k_t/m)\dot{y}_F \\ \cos(\varphi)\cos(\theta)(u_f/m) - g - (k_t/m)\dot{z}_F \\ -(k_r/I_x)p - qr(I_z - I_y)/I_x + (u_p/I_x) \\ -(k_r/I_y)q - pr(I_x - I_z)/I_y + (u_q/I_y) \\ -(k_r/I_z)r - pq(I_y - I_x)/I_z + (u_r/I_z) \\ p + q\sin(\varphi)\tan(\theta) + r\cos(\varphi)\tan(\theta) \\ q\cos(\varphi) - r\sin(\varphi) \\ (q\sin(\varphi) + r\cos(\varphi))/\cos(\theta) \end{bmatrix}.$$

Considerando dunque che il punto di equilibrio nel quale vogliamo linearizzare l'osservatore è un qualsiasi hover point dove i vettori x e u sono del tipo:

$$\begin{aligned} x_e &= \text{col}(x_{F0}, y_{F0}, z_{F0}, 0, 0, 0, 0, 0, 0, 0, 0, \psi_0) \\ u_e &= \text{col}(mg, 0, 0, 0). \end{aligned}$$

Allora è possibile calcolare le matrici A e B nel seguente modo:

$$A = \frac{\partial F(x_e, u_e)}{\partial x} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -\frac{k_t}{m} & 0 & 0 & 0 & 0 & 0 & g\sin(\psi_0) & g\cos(\psi_0) & 0 \\ 0 & 0 & 0 & 0 & -\frac{k_t}{m} & 0 & 0 & 0 & 0 & -g\cos(\psi_0) & g\sin(\psi_0) & 0 \\ 0 & 0 & 0 & 0 & 0 & -\frac{k_t}{m} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -\frac{k_r}{I_x} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\frac{k_r}{I_y} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\frac{k_r}{I_z} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

$$B = \frac{\partial F(x_e, u_e)}{\partial u} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \frac{1}{m} & 0 & 0 & 0 \\ 0 & \frac{1}{I_x} & 0 & 0 \\ 0 & 0 & \frac{1}{I_y} & 0 \\ 0 & 0 & 0 & \frac{1}{I_z} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

N.B: Nel nostro caso abbiamo scelto come punto di equilibrio il punto in cui il drone si trova dopo il primo takeoff 100 (dunque le componenti x_{F0}, y_{F0} e ψ_0 sono pari a 0 mentre z_{F0} è stato impostato a 100). Inoltre i coefficenti k_t e k_r sono stati posti momentaneamente a 0 poiché la velocità è molto bassa. Tutti i valori modificati sono costanti e dunque facilmente sostituibili nella nostra implementazione in C++.
In cima all'header sono definite tutte le costanti necessarie.

2.5 Codifica dell'osservatore

Per la codifica dell'osservatore è stato creato un nuovo header file all'interno della cartella **../libraries/SITL** chiamato `Observer.h` il quale successivamente sarà incluso nel file `SIM_Airsim.ccp` in modo che lo stato dell'osservatore sia calcolato per ogni frame della simulazione all'interno della funzione update.

2.5.1 Design e calcolo della matrice H

Al momento non esiste nessuna funzione in Boost, o in altre librerie, che permetta di calcolare la matrice H direttamente in C++, e scrivere manualmente l'algoritmo per il placement di autovalori sarebbe troppo oneroso e fuori obiettivo di questo progetto. Per questo al momento la matrice H è stata calcolata tramite *MatLab* grazie alla sua funzione `place`, e i valori sono riportati a mano nel codice C++.
Di seguito lo script MatLab per calcolare la matrice H.

```
%% Costanti
m = 1.22;
g = 9.807;
Ix = 0.008154;
```

```
Iy = 0.009846;
Iz = 0.017458;
phi_zero = 0.0;
kt = 0.0;
kr = 0.0;

%% MATRICI
A = [0 0 0 1 0 0 0 0 0 0 0 0
     0 0 0 0 1 0 0 0 0 0 0 0
     0 0 0 0 0 1 0 0 0 0 0 0
     0 0 0 -kt/m 0 0 0 0 0 g*sin(phi_zero) g*cos(phi_zero) 0
     0 0 0 0 -kt/m 0 0 0 0 -g*cos(phi_zero) g*sin(phi_zero) 0
     0 0 0 0 0 -kt/m 0 0 0 0 0 0
     0 0 0 0 0 0 -kr/Ix 0 0 0 0 0
     0 0 0 0 0 0 0 -kr/Iy 0 0 0 0
     0 0 0 0 0 0 0 0 -kr/Iz 0 0 0
     0 0 0 0 0 0 1 0 0 0 0 0
     0 0 0 0 0 0 0 1 0 0 0 0
     0 0 0 0 0 0 0 0 1 0 0 0];

I = eye(12, 12); % Matrice di identita'

% Inserimento di 12 autovalori (negativi)
H = place(A', I' ,[-23;-22;-21;-20;-19;-18;-17;-16;-15;-14;-13;-12]).';
writematrix(H) % Scrittura della matrice nel file H.txt
```

2.5.2 Inizializzazione e stima degli stati

Nel costruttore di AirSim vengono innanzitutto inizializzate tutte le variabili (matrici e vettori del punto di equilibrio) necessarie per l'osservatore.
Il sistema di equazioni differenziali ordinarie è definito nella funzione `observer_system`.

```
238     // Sistema di equazioni differenziali dell'osservatore
239     void observer_system(const state_type& x, state_type& dxdt, const double t)
240     {
241         dxdt = prod(A, x) + prod(B, delta_u) + prod(H, delta_x - x);
242     }
```

Figura 2.2: Sistema di ODE dell'osservatore

La funzione che invece effettua la stima vera e propria dello stato e quindi risolve il sistema di ODE è `observer`.

```
250     // Funzione dell'osservatore richiamata nella funzione update() di AirSim
251     void observer(uint64_t time, vector<double> prev_delta_obs_state, vector<double> sensor_state, const sitl_input& input)
252     {
253         delta_u = calc_input(input) - ue;
254         delta_x = sensor_state - xe;
255
256         integrate_adaptive(make_controlled(1E-12, 1E-12, stepper_type()), observer_system, prev_delta_obs_state, 0.0, 3.0E-3, 3.0E-3, save_obs_state);
257
258         vector<double> obs_state = delta_obs_state + xe;
259
260         saveToFile(time, obs_state, file1);
261         saveToFile(time, sensor_state, file2);
262         saveToFile(time, obs_state - sensor_state, file3);
263     }
```

Figura 2.3: Stima dello stato

Questa funzione riceve come parametri: il timestamp del frame corrente, lo stato stimato dall'osservatore del frame precedente (il delta), lo stato dei sensori e gli input PWM del frame corrente.
Con tali parametri prima vengono calcolati i delta del vettore di input e dello stato del sensore, sottraendo rispettivamente i punti di equilibrio u_e e x_e. Di seguito si richiama la funzione di Boost `integrate_adatpive` che risolve il sistema di ODE integrando dal frame precedente a quello corrente. Il risultato dell'osservatore viene salvato nel vettore globale `delta_obs_state` tramite la funzione `save_obs_state` passata come puntatore a `integrate_adaptive`.
Infine viene sommato il punto di equilibrio per ottenere lo stato effettivo e si salvano in tre file differenti di logging (`Sensor state.txt, Error.txt, Observer state.txt` in **ArduPilot/ArduCopter**) gli stati dell'osservatore, del sensore e l'errore. Di seguito un estratto da uno di questi file di logging:

```
1644230077096643 -1.66533e-16 1.33227e-15 1.82743e-13 0 0 -0.544833 -0.000527127
    0.00120822 0.00228804 1.41974e-05 -1.98547e-05 -1.71693e-05
1644230077099643 -1.66533e-16 1.33227e-15 1.82743e-13 0 0 -0.544833 0.00124864
    0.0017432 -0.00109257 1.05346e-05 -2.52849e-05 -1.39329e-05
1644230077102643 -1.66533e-16 1.33227e-15 1.82743e-13 0 0 -0.544833 0.00142764
    0.000896997 -0.00239656 6.27705e-06 -2.81474e-05 -6.6435e-06
1644230077105644 -1.66533e-16 1.33227e-15 1.82743e-13 0 0 -0.544833 0.000913068
    0.000111418 -0.000305502 3.56605e-06 -2.86051e-05 -5.73204e-06
1644230077108644 -1.66533e-16 1.33227e-15 1.82743e-13 0 0 -0.544833 -0.00102244
    0.00145107 0.00301579 6.80997e-06 -3.31969e-05 -1.50485e-05
```

La prima colonna è il timestamp del frame, dalla seconda in poi sono riportati in ordine $x, y, z, V_x, V_y, V_z, p, q, r, pitch, roll, yaw$.
La funzione `observer` viene richiamata nel metodo `update` di AirSim (`SITL_Airsim.cpp`).

```
427     // Wrapping dello stato in strutture dati custom e vettore boost
428     Geo home = { origin.lat / 1.0e7, origin.lng / 1.0e7, origin.alt / 100.0f };
429     Geo geo = { state.gps.lat, state.gps.lon, state.gps.alt };
430     vector<double> pos(3);
431     pos = GeodeticToNed(geo, home);
432
433     vector<double> sensor_state(12);
434     sensor_state(0) = pos(0);
435     sensor_state(1) = pos(1);
436     sensor_state(2) = state.gps.alt;
437     sensor_state(3) = state.velocity.world_linear_velocity.x;
438     sensor_state(4) = state.velocity.world_linear_velocity.y;
439     sensor_state(5) = state.velocity.world_linear_velocity.z;
440     sensor_state(6) = state.imu.angular_velocity.x;
441     sensor_state(7) = state.imu.angular_velocity.y;
442     sensor_state(8) = state.imu.angular_velocity.z;
443     sensor_state(9) = state.pose.pitch;
444     sensor_state(10) = state.pose.roll;
445     sensor_state(11) = state.pose.yaw;
446
447     // Controllo dello stato per evitare eccezioni dovute alla divergenza dell'osservatore all'inizio della simulazione e in casistiche estreme
448     if ((sensor_state(0) != 0.0f || sensor_state(1) != 0.0f || sensor_state(2) != 0.0f) &&
449         (sensor_state(0) < 10000.0f && sensor_state(1) < 10000.0f && sensor_state(2) < 10000.0f) &&
450         (sensor_state(0) > -10000.0f && sensor_state(1) > -10000.0f && sensor_state(2) > -10000.0f))
451     {
452         observer(state.timestamp, prev_delta_obs_state, sensor_state, input);
453         prev_delta_obs_state = delta_obs_state;
454     }
```

Figura 2.4: Richiamo di observer()

Prima vengono convertite le coordinate Geospaziali in coordinate NED, scalando adeguatamente le coordinate del punto di origine. Di seguito si effettua un wrap-

ping dello stato del sensore in un vettore Boost. Infine si richiama observer solo dopo aver verificato che le componenti della posizione non siano nulle o abbiano valori elevati, i valori nulli indicano che il drone non è ancora accesso, i valori troppo elevati invece sono dovuti ad alcune divergenze temporanee in fase di accensione che potrebbero però mandare in errore l'osservatore e bloccare la simulazione.

Capitolo 3

Risultati e test

3.1 Generazione dei risultati

I file .txt generati durante i test sono stati utilizzati in un notebook di Google Colab per la costruzione di due grafici uno relativo all'errore della stima dell'osservatore e uno relativo al confronto tra la funzione di stima dell'osservatore e lo stato dei sensori. Il notebook è disponibile al seguente link:

`https://colab.research.google.com/drive/15lxL61HogjzJTFgzB0akkutbHkIgZENt?usp=sharing`

N.B. Il file notebook nel caso di test con fault prende in input i file "Error con fault.txt", "Sensor State con fault.txt" e "Observer State con fault.txt" dunque è necessario rinominare i file generati.

3.2 No Fault

Il primo test è stato effettuato senza inserire alcun tipo di fault in modo da verificare il corretto funzionamento dell'osservatore. Dunque sarà sufficente avviare una mission come descritto nei paragrafi precedenti e graficare i risultati ottenuti. É possibile notare come l'errore dei vari stati del sistema oscilli sempre sullo 0 (dunque la stima dell'osservatore è conforme all'evoluzione dello stato rilevato dai sensori).

Figura 3.1: Errori dell'osservatore senza fault

Figura 3.2: Confronto tra lo stato dell'osservatore e quello del sensore senza fault

3.3 Fault GPS

In questo test è stato iniettato un fault nelle tre componenti del GPS tramite il comando `SIM_GPS_GLITCH_POS` dove pos è una delle tre componenti X, Y e Z. Il fault aggiunge al sensore ogni 10ms il valore settato in metri. In questo caso si può notare come i valori di x e y decrescano improvvisamente e come l'errore in x, y, p, q oscilli maggiormente rispetto al caso senza fault. In particolare i valori dell'errore in p e q oscillerano oltre lo 0 in un intervallo [-2, 2].

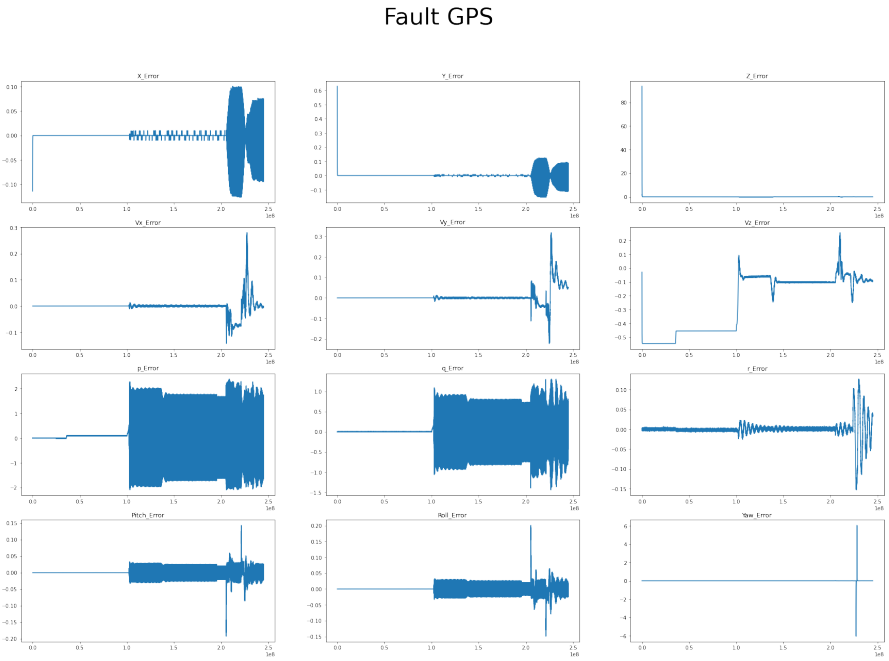


Figura 3.3: Errori dell'osservatore con fault sul GPS

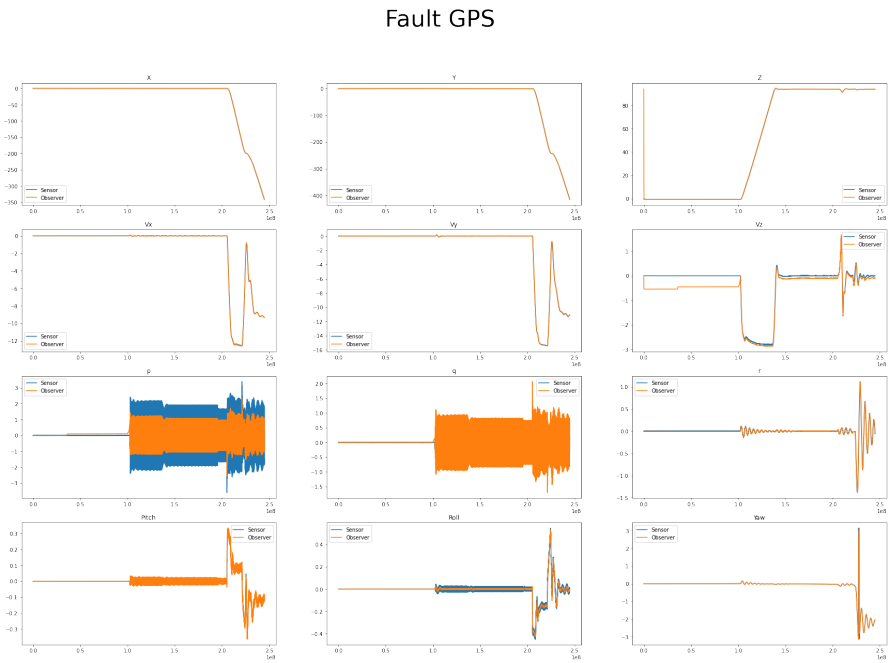


Figura 3.4: Confronto tra lo stato dell'osservatore e quello del sensore con fault sul GPS

3.4 Fault accelerometro

In questo test sono stati iniettati dei fault negli accelerometri del quadrirotore tramite il comando `SIM_ACC_BIAS_POS` e `SIM_ACC2_BIAS_POS` dove pos è una delle componenti X, Y e Z. Il fault aggiunge del rumore nella componente specificata, maggiore è il valore iniettato e maggiore sarà il rumore nel sensore. In maniera analoga al fault sul GPS è possibile notare come i valori di x e y decrescano (in maniera minore) e gli errori di p e q oscillino in un intervallo [-2, 2].

Figura 3.5: Errori dell'osservatore con fault sui due accelerometri

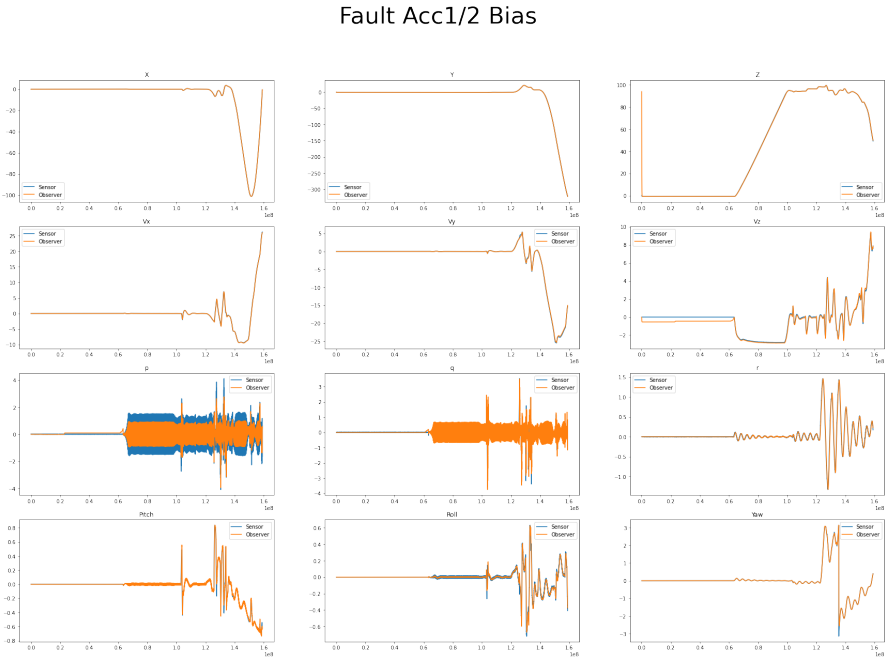


Figura 3.6: Confronto tra lo stato dell'osservatore e quello del sensore con fault sui due accelerometri

3.5 Noise giroscopio

In questo test è stato iniettato del rumore al giroscopio tramite il comando `SIM_GYR_RND_POS` dove pos è une delle componenti X, Y e Z. Maggiore è il valore impostato maggiore sarà il rumore iniettato al sensore. In questo caso le componenti *pitch* e *roll* oscillano con maggiore frequenza, inoltre la componente y decresce in maniera più pronunciata rispetto ad x. Infine è possibile notare un discostamento dallo 0 nelle tre componenti delle velocità V_x, V_y, V_z ed un' oscillazione di q tra [-4, 4] e p nell'intervallo [-2, 2].

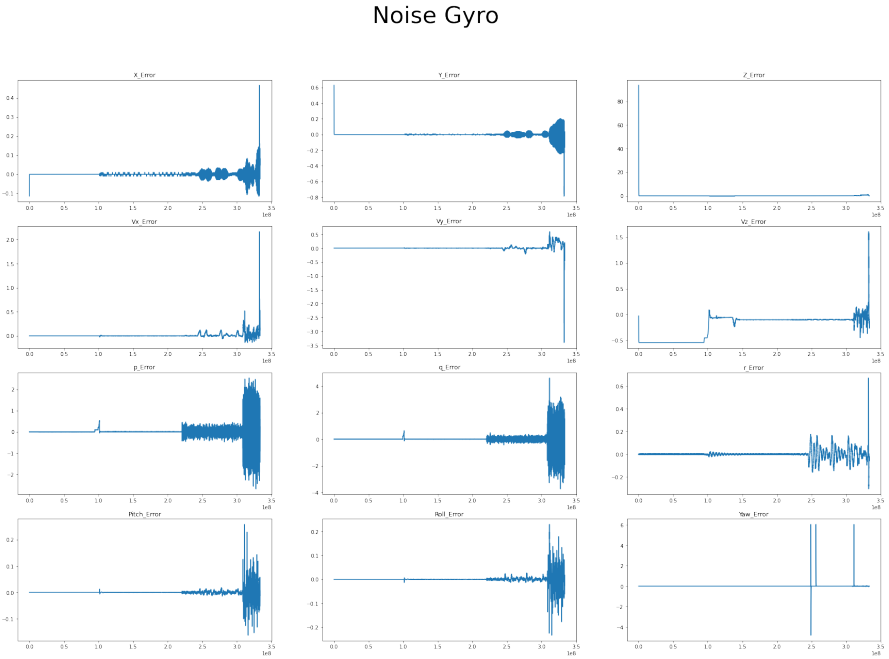


Figura 3.7: Errori dell'osservatore con rumore sul giroscopio

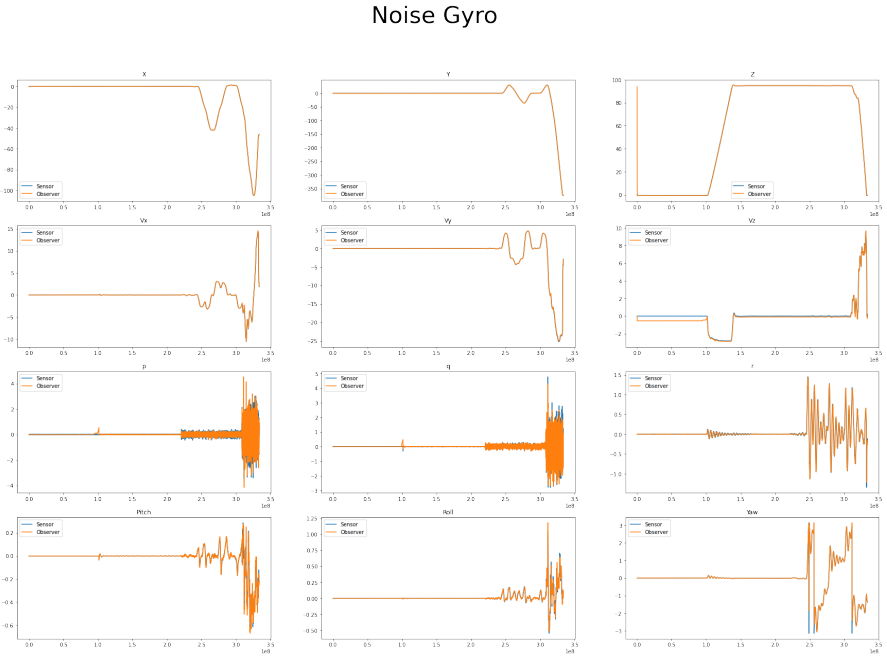


Figura 3.8: Confronto tra lo stato dell'osservatore e quello del sensore con rumore sul giroscopio

3.6 Fault barometro

In questo test è stato iniettato un fault al barometro del drone tramite il comando `SIM_BARO_DRIFT` il quale deriva in m/s il barometro con un effetto che porta al drone a salire e scendere di un determinato valore. In questo caso è possibile notare una leggera oscillazione di p e q nell' intervallo [-1, 1].

Figura 3.9: Errori dell'osservatore con fault sul barometro

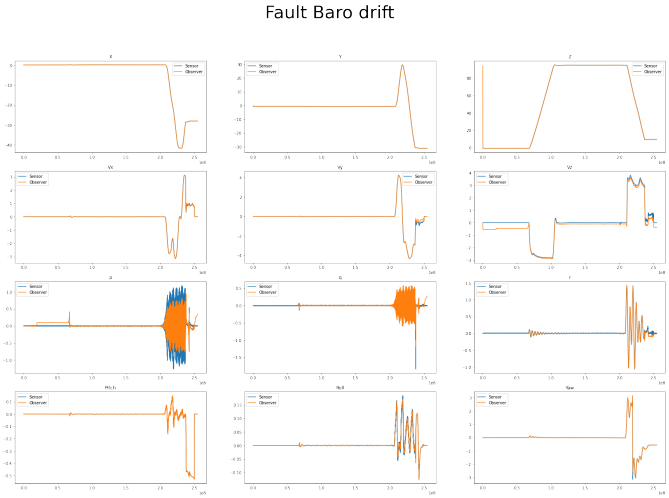


Figura 3.10: Confronto tra lo stato dell'osservatore e quello del sensore con fault sul barometro

N.B. Tutti i fault testati sono stati iniettati dopo il takeoff 100 e con valori sempre più crescenti durante lo svolgimento della mission. Inoltre una volta finito ogni test è necessario reimpostare i valori dei fault al valore di default (solitamente pari a 0) per un corretto funzionamento del drone.