# Analisi del `Controller::compute()`

Controllore a cascata per quadricottero in hovering

Computational Modeling for Robotics and HCI — University of Messina

Dinamica e Controllo degli UAV

**Obiettivo del documento**

Questo documento analizza riga per riga la funzione `Controller::compute()` che implementa il controllore in cascata a tre livelli per un quadricottero. Per ogni blocco di codice vengono mostrate: la matematica sottostante, la motivazione fisica e la connessione con le equazioni del moto del capitolo 5. In fondo al documento sono proposti esercizi guidati.

## 1 Il codice completo

Listing 1: Controller::compute() — controller.cpp

```
void Controller::compute(const mjModel* m, mjData* d)
{
    (void)m;
    const double* pos  = d->qpos;        // [x, y, z]
    const double* quat = d->qpos + 3;    // [w, x, y, z]
    const double* vel  = d->qvel;        // velocita' in mondo
    const double* omg  = d->qvel + 3;    // omega in mondo

    // --- Orientazione ---
    double q[4] = {quat[0], quat[1], quat[2], quat[3]};
    qmath::normalize(q);
    double roll, pitch, yaw;
    qmath::to_euler(q, roll, pitch, yaw);

    // omega nel frame body: omega_b = R^T * omega_world
    double R[3][3];
    qmath::to_rotmat(q, R);
    double omg_b[3] = {0, 0, 0};
    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
            omg_b[i] += R[j][i] * omg[j];    // R[j][i] = R^T[i][j]

    // --- Loop altitudine ---
    double f_total = DRONE_MASS * (GRAVITY
                   + Kp_z * (z_des - pos[2])
                   + Kd_z * (-vel[2]));
    f_total = std::max(0.0, f_total);

    // --- Loop posizione x/y ---
    const double ax = Kp_xy*(x_des - pos[0]) + Kd_xy*(-vel[0]);
    const double ay = Kp_xy*(y_des - pos[1]) + Kd_xy*(-vel[1]);
    const double theta_des = std::clamp( ax/GRAVITY, -max_tilt, max_tilt
        );
    const double phi_des   = std::clamp(-ay/GRAVITY, -max_tilt, max_tilt
        );

```

```
35      // --- Loop assetto ---
36      const double tau_x = Kp_roll  * (phi_des   - roll)               -
            Kd_roll  * omg_b[0];
37      const double tau_y = Kp_pitch * (theta_des - pitch)              -
            Kd_pitch * omg_b[1];
38      const double tau_z = Kp_yaw   * wrap_pi(yaw_des - yaw)           -
            Kd_yaw   * omg_b[2];
39
40      // --- Mixing ---
41      const double l = ARM_LENGTH, c = KM_KF;
42      const double T4 = f_total * 0.25;
43      auto sat = [](double v){ return std::clamp(v, 0.0, MAX_THRUST); };
44      d->ctrl[0] = sat(T4 - tau_z/(4*c) - tau_y/(2*l));  // front CW
45      d->ctrl[1] = sat(T4 + tau_z/(4*c) + tau_x/(2*l));  // left  CCW
46      d->ctrl[2] = sat(T4 - tau_z/(4*c) + tau_y/(2*l));  // back  CW
47      d->ctrl[3] = sat(T4 + tau_z/(4*c) - tau_x/(2*l));  // right CCW
48  }
```

## 2 Struttura dell'interfaccia con MuJoCo

La funzione riceve due puntatori: `m` (modello, costante, contiene parametri fisici e topologia) e `d` (dati, aggiornato ad ogni step).

**Layout di `d->qpos` e `d->qvel` per un `freejoint`**

| Indici | Variabile | Significato |
|---|---|---|
| `qpos[0:3]` | $x, y, z$ | posizione nel frame mondo [m] |
| `qpos[3:7]` | $w, x_q, y_q, z_q$ | quaternione di orientazione |
| `qvel[0:3]` | $\dot{x}, \dot{y}, \dot{z}$ | velocità lineare, frame mondo [m/s] |
| `qvel[3:6]` | $\omega_x, \omega_y, \omega_z$ | velocità angolare, frame mondo [rad/s] |

Il vettore `ctrl[0..3]` è l'unico canale di comando: MuJoCo lo legge all'inizio di ogni `mj_step()` e lo converte in forze/coppie tramite la sezione `<actuator>` del file XML.

## 3 Blocco 1 — Lettura dello stato e geometria del quaternione

Listing 2: Lettura stato e normalizzazione

```
9       double q[4] = {quat[0], quat[1], quat[2], quat[3]};
10      qmath::normalize(q);
11      double roll, pitch, yaw;
12      qmath::to_euler(q, roll, pitch, yaw);
```

### Perché normalizzare il quaternione?

Un quaternione unitario $\boldsymbol{q} = (w, x, y, z)$ con $\|\boldsymbol{q}\| = 1$ rappresenta un'orientazione pura su $SO(3)$. MuJoCo mantiene la norma quasi unitaria durante l'integrazione, ma piccoli errori numerici si accumulano; la normalizzazione garantisce che la conversione agli angoli di Eulero sia sempre valida.

### Formula di conversione quaternione → Eulero ZYX

La convenzione ZYX (roll–pitch–yaw) decompone la rotazione come:

$$\boldsymbol{R} = R_z(\psi) \cdot R_y(\theta) \cdot R_x(\phi)$$

I tre angoli si estraggono da $\boldsymbol{q} = (w, x, y, z)$ con:

**Euler ZYX da quaternione**

$$\phi = \operatorname{atan2}\big(2(wx + yz),\ 1 - 2(x^2 + y^2)\big) \tag{1}$$

$$\theta = \arcsin\big(2(wy - zx)\big) \tag{2}$$

$$\psi = \operatorname{atan2}\big(2(wz + xy),\ 1 - 2(y^2 + z^2)\big) \tag{3}$$

**Singolarità di gimbal lock**

La formula per $\theta$ usa arcsin, che ha una singolarità a $|\sin\theta| = 1$, cioè $\theta = \pm 90$. In quella configurazione il sistema perde un grado di libertà (roll e yaw diventano indistinguibili). Per questo motivo in sistemi che eseguono manovre acrobatiche si usa il controllo geometrico diretto su $SO(3)$ invece degli angoli di Eulero.

## 4 Blocco 2 — Velocità angolare nel frame body

Listing 3: Rotazione di omega dal frame mondo al frame body

```
14      double R[3][3];
15      qmath::to_rotmat(q, R);
16      double omg_b[3] = {0, 0, 0};
17      for (int i = 0; i < 3; i++)
18          for (int j = 0; j < 3; j++)
19              omg_b[i] += R[j][i] * omg[j];    // R[j][i] = R^T[i][j]
```

### Perché questa operazione è necessaria?

MuJoCo memorizza la velocità angolare nel **frame mondo**: $\boldsymbol{\omega}^W \in \mathbb{R}^3$. Il controllore PD di assetto ha bisogno delle componenti $(p, q, r)$ nel **frame body** per calcolare il termine derivativo correttamente. La relazione tra i due frame è:

$$\boxed{\boldsymbol{\omega}^B = \boldsymbol{R}^T \boldsymbol{\omega}^W}$$

dove $\boldsymbol{R}$ è la matrice di rotazione *da body a mondo*.

### Connessione con l'implementazione

Il doppio ciclo `omg_b[i] += R[j][i] * omg[j]` calcola proprio $\boldsymbol{R}^T\boldsymbol{\omega}^W$: accedere a `R[j][i]` invece di `R[i][j]` equivale a usare la trasposta senza allocare una matrice aggiuntiva.

**Verifica dimensionale**

$\boldsymbol{R} \in \mathbb{R}^{3\times3}$ (ortogonale, $\boldsymbol{R}^T = \boldsymbol{R}^{-1}$), $\boldsymbol{\omega}^W \in \mathbb{R}^3$, quindi $\boldsymbol{R}^T\boldsymbol{\omega}^W \in \mathbb{R}^3 = \boldsymbol{\omega}^B$ ✓

# 5 Blocco 3 — Loop di altitudine

Listing 4: Controllo PD di quota

```
21    double f_total = DRONE_MASS * (GRAVITY
22                   + Kp_z * (z_des - pos[2])
23                   + Kd_z * (-vel[2]));
24    f_total = std::max(0.0, f_total);
```

### Derivazione dalla dinamica

L'equazione del moto verticale per piccoli angoli è:

$$m\ddot{z} = f\cos\phi\cos\theta - mg \approx f - mg$$

Imponiamo una legge PD sul secondo membro:

$$m\ddot{z} = K_{p,z}(z_d - z) + K_{d,z}(0 - \dot{z})$$

Sommando la compensazione della gravità (feedforward):

$$\boxed{f = m\Big(g + K_{p,z}(z_d - z) - K_{d,z}\dot{z}\Big)}$$

Questa è esattamente la riga di codice, con `pos[2]` $= z$ e `vel[2]` $= \dot{z}$.

**Proprietà a regime**

In steady state ($\ddot{z} = \dot{z} = 0$, $z = z_d$):

$$f^* = mg = 0.027 \times 9.81 \approx 0.265 \text{ N}$$

Ogni rotore eroga $f^*/4 \approx 0.066$ N, pari al 44% della sua spinta massima (`MAX_THRUST` $= 0.15$ N).

La chiamata a `std::max(0.0, f_total)` impone il **vincolo fisico**: i rotori non possono generare spinta negativa.

# 6 Blocco 4 — Loop di posizione x/y

Listing 5: Outer-outer loop: posizione orizzontale

```
25    const double ax = Kp_xy*(x_des - pos[0]) + Kd_xy*(-vel[0]);
26    const double ay = Kp_xy*(y_des - pos[1]) + Kd_xy*(-vel[1]);
27    const double theta_des = std::clamp( ax/GRAVITY, -max_tilt, max_tilt
          );
28    const double phi_des   = std::clamp(-ay/GRAVITY, -max_tilt, max_tilt
          );
```

### La linearizzazione di piccoli angoli

Le equazioni del moto orizzontali complete sono:

$$m\ddot{x} = f\left(c_\psi s_\theta c_\phi + s_\psi s_\phi\right) \tag{4}$$

$$m\ddot{y} = f\left(s_\psi s_\theta c_\phi - c_\psi s_\phi\right) \tag{5}$$

Per $\phi \approx 0$, $\theta \approx 0$, $\psi \approx 0$ e $f \approx mg$:

$$m\ddot{x} \approx f\,\theta \approx mg\,\theta \qquad \Rightarrow \qquad \ddot{x} \approx g\,\theta$$

$$m\ddot{y} \approx -f\,\phi \approx -mg\,\phi \qquad \Rightarrow \qquad \ddot{y} \approx -g\,\phi$$

Invertendo, i setpoint di assetto necessari per ottenere un'accelerazione orizzontale $(a_x, a_y)$ sono:

$$\boxed{\theta_d = \frac{a_x}{g}, \qquad \phi_d = -\frac{a_y}{g}}$$

dove $a_x, a_y$ sono le uscite del PD di posizione:

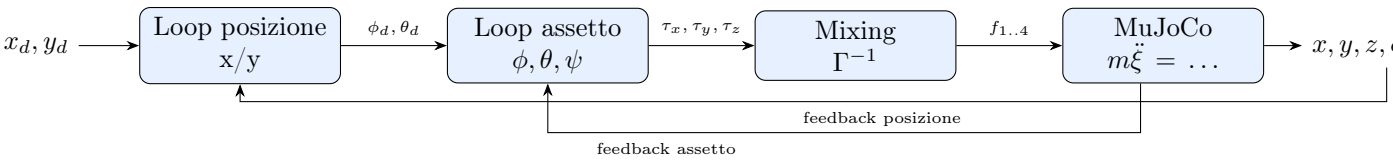
$$a_x = K_{p,xy}(x_d - x) + K_{d,xy}(0 - \dot{x})$$

**Limite di inclinazione `max_tilt`**

`std::clamp` limita i setpoint di assetto a $\pm$`max_tilt` (default: 0.3 rad $\approx$ 17). Questo è fondamentale per due ragioni:

- La linearizzazione $\ddot{x} \approx g\theta$ cessa di essere valida per angoli grandi.
- Grandi inclinazioni riducono la spinta verticale efficace $f\cos\theta$, compromettendo il controllo di quota.

### Schema della cascata outer-outer → outer → inner




## 7 Blocco 5 — Loop di assetto (inner loop)

Listing 6: Controllo PD di assetto

```
30    const double tau_x = Kp_roll  * (phi_des   - roll)            -
         Kd_roll  * omg_b[0];
31    const double tau_y = Kp_pitch * (theta_des - pitch)           -
         Kd_pitch * omg_b[1];
32    const double tau_z = Kp_yaw   * wrap_pi(yaw_des - yaw)        -
         Kd_yaw   * omg_b[2];
```

Le tre coppie sono calcolate con la stessa struttura PD:

$$\boxed{\tau_i = K_{p,i}\,e_i - K_{d,i}\,\omega_{b,i}}$$

dove $e_i$ è l'errore angolare e $\omega_{b,i}$ è la velocità angolare nel frame body (calcolata nel Blocco 2).

**Perché `wrap_pi` per lo yaw?**

Lo yaw $\psi$ è un angolo che vive in $(-\pi, \pi]$. Senza `wrap_pi`, l'errore $e_\psi = \psi_d - \psi$ potrebbe valere $+\pi$ o $-\pi$ per la stessa configurazione fisica (il drone è allo stesso orientamento

visto da angoli opposti). `wrap_pi` riporta l'errore nel range $(-\pi, \pi)$, garantendo che il controllore scelga sempre il percorso più breve. Esempio: se $\psi_d = 170$ e $\psi = -170$, l'errore "grezzo" sarebbe 340 (giro completo), mentre `wrap_pi` lo corregge a $-20$.

Il termine derivativo usa $\boldsymbol{\omega}^B$ e non $\dot{\boldsymbol{\eta}}$ (derivata degli angoli di Eulero). I due coincidono solo per angoli piccoli; usare $\boldsymbol{\omega}^B$ è fisicamente corretto in tutte le condizioni.

# 8 Blocco 6 — Mixing (inversione di Γ)

Listing 7: Allocazione inversa delle forze ai rotori

```
35    const double l = ARM_LENGTH, c = KM_KF;
36    const double T4 = f_total * 0.25;
37    auto sat = [](double v){ return std::clamp(v, 0.0, MAX_THRUST); };
38    d->ctrl[0] = sat(T4 - tau_z/(4*c) - tau_y/(2*l));  // front CW
39    d->ctrl[1] = sat(T4 + tau_z/(4*c) + tau_x/(2*l));  // left  CCW
40    d->ctrl[2] = sat(T4 - tau_z/(4*c) + tau_y/(2*l));  // back  CW
41    d->ctrl[3] = sat(T4 + tau_z/(4*c) - tau_x/(2*l));  // right CCW
```

### Derivazione della matrice di allocazione

Dalla definizione di spinta ($f_i = k_f \Omega_i^2$) e coppia di drag ($\tau_i = k_m \Omega_i^2$), con $c = k_m/k_f$ e braccio $l$:

$$f_{\text{tot}} = f_1 + f_2 + f_3 + f_4 \tag{6}$$
$$\tau_\phi = l\,(f_2 - f_4) \tag{7}$$
$$\tau_\theta = l\,(f_3 - f_1) \tag{8}$$
$$\tau_\psi = c\,(-f_1 + f_2 - f_3 + f_4) \tag{9}$$

In forma matriciale $\boldsymbol{u} = \Gamma\,\boldsymbol{f}$:

$$\underbrace{\begin{pmatrix} f \\ \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{pmatrix}}_{\boldsymbol{u}} = \underbrace{\begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & l & 0 & -l \\ -l & 0 & l & 0 \\ -c & c & -c & c \end{bmatrix}}_{\Gamma} \underbrace{\begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \end{pmatrix}}_{\boldsymbol{f}}$$

Invertendo analiticamente Γ si ottiene:

$$\begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \end{pmatrix} = \begin{bmatrix} \frac{1}{4} & 0 & -\frac{1}{2l} & -\frac{1}{4c} \\ \frac{1}{4} & \frac{1}{2l} & 0 & \frac{1}{4c} \\ \frac{1}{4} & 0 & \frac{1}{2l} & -\frac{1}{4c} \\ \frac{1}{4} & -\frac{1}{2l} & 0 & \frac{1}{4c} \end{bmatrix} \begin{pmatrix} f \\ \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{pmatrix}$$

Le quattro righe corrispondono esattamente alle quattro assegnazioni di `ctrl[0..3]` nel codice.

**La lambda `sat`**

In C++14 e successivi, una funzione locale può essere definita inline come *lambda*:

```
auto sat = [](double v){ return std::clamp(v, 0.0, MAX_THRUST); };
```

È equivalente a una funzione statica ma visibile solo localmente. Impone il vincolo fisico: le forze dei rotori devono essere $\in [0, \texttt{MAX_THRUST}]$.

# 9 Parametri fisici e guadagni

| Parametro | Valore | Unità | Ruolo |
|---|---|---|---|
| `GRAVITY` | 9.81 | $m/s^2$ | Compensazione feedforward |
| `DRONE_MASS` | 0.027 | kg | Massa totale Crazyflie 2.x |
| `ARM_LENGTH` | 0.046 | m | Braccio $l$ per mixing |
| `KM_KF` | 0.016 | — | Rapporto $k_m/k_f$ |
| `MAX_THRUST` | 0.15 | N | Saturazione per rotore |
| `Kp_z` | 12.0 | N/m | Stiffness quota |
| `Kd_z` | 5.0 | N·s/m | Damping quota |
| `Kp_roll` | $8 \times 10^{-4}$ | N·m/rad | Stiffness roll |
| `Kd_roll` | $2 \times 10^{-4}$ | N·m·s/rad | Damping roll |
| `Kp_xy` | 2.5 | N/m | Stiffness posizione xy |
| `Kd_xy` | 2.0 | N·s/m | Damping posizione xy |
| `max_tilt` | 0.3 | rad ($\approx 17$) | Limite inclinazione |

### Regola di smorzamento critico

Per il loop di altitudine, smorzamento critico si ottiene con:

$$K_{d,z} = 2\sqrt{K_{p,z} \cdot m} = 2\sqrt{12.0 \times 0.027} \approx 1.14 \text{ N·s/m}$$

Il valore usato ($K_{d,z} = 5.0$) è maggiore $\Rightarrow$ il sistema è **sovrasmorzato** in quota (risposta lenta ma senza oscillazioni).

# 10 Esercizi

### Esercizio 1 — Analisi del quaternione (6 pt)

Il drone parte con la configurazione `qpos[3:7] = [1, 0, 0, 0]` (quaternione identità).

a) Quali sono i valori di roll, pitch, yaw restituiti da `qmath::to_euler()`?

b) Se si sostituisce il quaternione con $\boldsymbol{q} = (0.966, 0.259, 0, 0)$ (corrispondente a roll = 30), verifica la formula (1) manualmente con calcolatrice.

c) Cosa restituisce `to_euler` per $\boldsymbol{q} = (0.707, 0, 0.707, 0)$? Qual è l'angolo di pitch corrispondente? Sei vicino alla singolarità? Motiva la risposta.

### Esercizio 2 — Rotazione di $\boldsymbol{\omega}$ (6 pt)

Il drone ha velocità angolare *nel frame mondo* $\boldsymbol{\omega}^W = (0,\ 0,\ 1)^T$ rad/s (rotazione intorno all'asse $z$ verticale). L'orientazione del body è descritta da $\boldsymbol{R} = \mathbf{I}$ (identità).

a) Calcola $\boldsymbol{\omega}^B = \boldsymbol{R}^T\boldsymbol{\omega}^W$ a mano. Risultato atteso: $\boldsymbol{\omega}^B =$?

b) Ripeti il calcolo con $\boldsymbol{R} = R_x(90)$ (body inclinato di 90° in roll). Cosa osservi sulla componente $r = \omega_{b,2}$?

c) Nel codice il ciclo usa `R[j][i]` invece di `R[i][j]`: spiega perché questo equivale a moltiplicare per la trasposta senza creare una copia di $\boldsymbol{R}$.

### Esercizio 3 — Loop di altitudine (8 pt)

a) Il drone è a $z = 0.3$ m, velocità verticale nulla, setpoint $z_d = 0.5$ m. Calcola `f_total`.

b) Cosa succederebbe se si rimuovesse il termine `GRAVITY` dall'espressione? Descrivi il comportamento atteso del drone.

c) Con $K_{p,z} = 12$, $K_{d,z} = 5$, $m = 0.027$ kg, verifica se il sistema è sottosmorzato, criticamente smorzato o sovrasmorzato usando la formula $\zeta = K_{d,z}/(2\sqrt{K_{p,z} \cdot m})$.

d) Proponi un valore di $K_{d,z}$ che dia smorzamento critico.

### Esercizio 4 — Linearizzazione e loop x/y (8 pt)

a) Partendo dalla forma completa di $\ddot{x}$:

$$\ddot{x} = \frac{f}{m}(c_\psi s_\theta c_\phi + s_\psi s_\phi)$$

mostra analiticamente che, per $\phi, \theta, \psi \to 0$ e $f \to mg$, si ottiene $\ddot{x} \approx g\theta$.

b) Il drone è in $x = 0.8$ m, $\dot{x} = 0.2$ m/s, setpoint $x_d = 0$, con $K_{p,xy} = 2.5$, $K_{d,xy} = 2.0$. Calcola $a_x$ e il corrispondente $\theta_d$ (in gradi). Supera `max_tilt`? Se sì, qual è il valore saturato?

c) Perché il segno di $\phi_d$ è opposto a quello di $\theta_d$ rispetto all'errore di posizione? Aiutati disegnando la geometria del drone visto dall'alto.

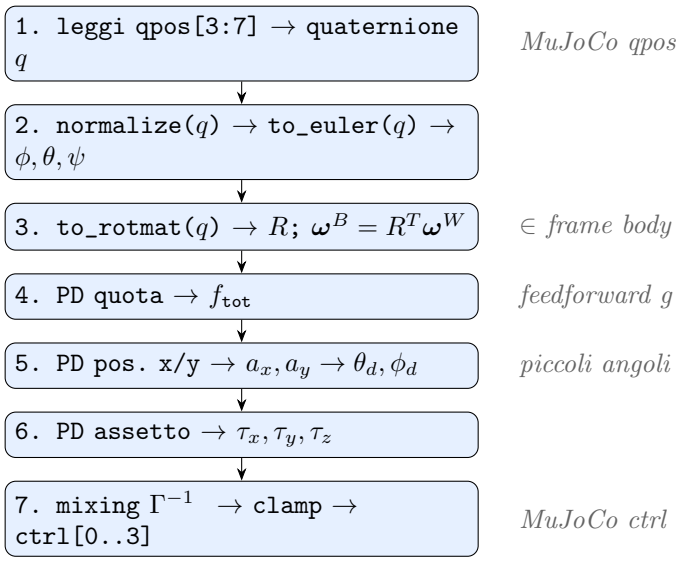
### Esercizio 5 — Mixing e matrice $\Gamma$ (12 pt)

a) Scrivi la matrice $\Gamma$ esplicita per la configurazione "+" con $l = 0.046$ m e $c = 0.016$.
b) Verifica che $\Gamma$ sia invertibile calcolando $\det(\Gamma)$ (può essere fatto sviluppando per colonne).
c) Hover puro: $f = mg = 0.265$ N, $\tau_\phi = \tau_\theta = \tau_\psi = 0$. Calcola $f_1, f_2, f_3, f_4$ usando le formule di mixing. Il risultato è fisicamente sensato?
d) Ora aggiungi una coppia di roll $\tau_\phi = 5 \times 10^{-5}$ N·m mantenendo hover in quota. Calcola i nuovi $f_1, f_2, f_3, f_4$. Quali rotori aumentano la spinta e quali la diminuiscono? È coerente con la geometria?

### Esercizio 6 — Domanda aperta: disturbo esterno (10 pt)

Un vento costante in direzione $x$ esercita una forza $F_d = 0.005$ N sul drone.
a) Con il controllore attuale (solo PD), il drone converge alla posizione desiderata $x_d = 0$, oppure si assesta con un errore stazionario? Giustifica analiticamente usando le equazioni del loop x/y.
b) Proponi una modifica al codice del loop di posizione per eliminare l'errore stazionario. Scrivi la formula e indica dove va inserita nella funzione `compute()`.
c) Quali precauzioni bisogna prendere con il termine integrativo nei sistemi di controllo di volo? (Suggerimento: considera cosa succede se il drone è bloccato a terra e il pilota tiene lo stick sul massimo per 10 secondi.)

## Riepilogo: flusso di esecuzione



**Messagio chiave**

Il codice non contiene fisica: la dinamica è interamente delegata a `mj_step()`. `Controller::compute()` implementa solo la **legge di controllo**: legge lo stato, calcola la risposta algebrica e scrive i comandi. La separazione tra controllore e simulatore è il principio architetturale fondamentale del progetto.