Stabilizzazione del pendolo inverso (con modello hardware)

Americo Cherubini

***Abstract*— Realizzazione e stabilizzazione di un modellino hardware del pendolo inverso.**

***Index Terms*— Enter key words or phrases in alphabetical order, separated by commas.**

I. Descrizione del progetto

Il pendolo inverso è uno dei sistemi instabili più gettonati per i ricercatori dei sistemi di controllo, dato il suo alto grado di instabilità naturale; riuscire a stabilizzarlo può essere considerata una sfida di ”hello world” nel mondo dei controlli, di complessità non affatto banale. Ci sono molti approcci che consentono l'adempimento di questo obiettivo. In questo report ci si focalizzerà sull'utilizzo dei controllori PID, una delle tipologie di controllori più utilizzati nel mondo della ricerca e dell'industria e uno dei più facili da implementare su microcontrollori simili ad Arduino.

II. Descrizione setup

Il modello del pendolo inverso utilizzato in questo progetto consiste in due unità hardware distinte:

- Controllore: si occupa dell'esecuzione degli algoritmi di controllo (PID) e di pilotaggio del motore passo passo;
- Lettore dell'angolo: montato sul carrello del pendolo, acquisisce dati sullo stato del pendolo e li trasmette al controllore.

A. Controllore

Il controllore consiste in una scheda ESP8266 (modello Wemos mini), alimentato direttamente dal cavo USB C dal PC. A bordo di questo microcontrollore viene eseguito il codice che ha il compito di ricevere i dati sullo stato del pendolo, calcolare l'accelerazione (input) da imprimere al carrello per la stabilizzazione, e comandare il motore passo con l'accelerazione calcolata. Il segnale di comando generato dal microcontrollore per pilotare il motore consiste in un treno di impulsi che viene mandato direttamente in ingresso al piedino “STEP” del driver del motore. Il driver è un A4988, ed è in grado di controllare un motore passo-passo bipolare a 4 fili con vari livelli di microstepping. L'alimentazione del motore è fornita da una batteria LiPo da 7.4v, connessa direttamente ai rispettivi piedini VMOT e GND del driver A4988. L'intero modulo è inscatolato in un case stampato in 3D, in cui alloggia la batteria e la basetta millefori su cui è stato realizzato il circuito. Dal driver escono i 4 fili delle due fasi del motore passo-passo, in cui il driver eroga la corrente necessaria per effettuare lo stepping. Il motore è fissato su un banco piano usando del nastro adesivo. *Fig. 2*

B. Carrello e lettore

Sul rotore del motore è fissato con una vite il blocco “carrello”, stampato in 3D. In esso alloggiano due cuscinetti radiali in cui è inserita una barra libera di ruotare. Su un'estremità della barra è bullonata l'asta del pendolo; sull'altra estremità è incollato un magnete con magnetizzazione diametrale, che ruota in prossimità dell'encoder rotatorio AS5600, consentendogli di rilevare l'orientamento dell'asta. Il microcontrollore utilizzato è un altro ESP8266 (Wemos mini), collocato su una piccola breadboard. Sulla breadboard è anche inserito un potenziometro, il cui segnale analogico viene acquisito dall'unico piedino ADC presente sulla scheda. Il codice a bordo del microcontrollore si occupa dell'acquisizione (tramite protocollo I2C) dell'angolo rilevato dall'encoder, lieve post-processamento dei dati (per mitigare il rumore e aggiungere un offset, regolabile tramite potenziometro), e trasmissione dei dati alla scheda ricevente utilizzando il protocollo wireless proprietario di Espressif ESP-NOW. *Fig. 1*

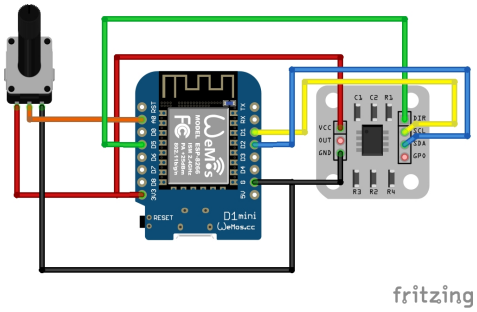


Fig. 1: Diagramma circuitale dell'unità lettore.

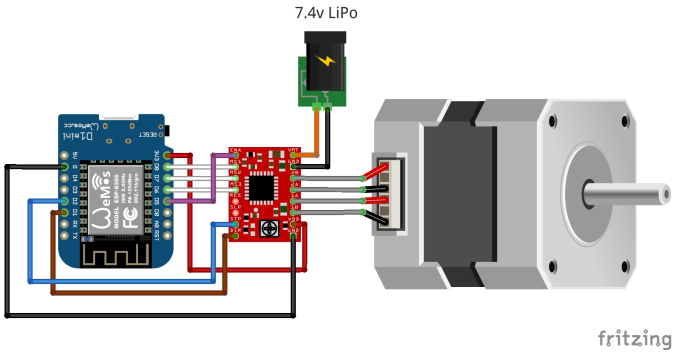


Fig. 2: Diagramma circuitale dell'unità controllore.

C. Strumenti software impiegati

Arduino IDE: per scrivere e caricare il codice sui microcontrollori, e per interagire tramite comandi su seriale.

III. Funzionamento del software embedded

Un'apposita libreria permette al codice in esecuzione sul MCU di lettura di acquisire il valore dell'angolo dal sensore magnetico. Nel `setup()`, il sensore viene letto una prima volta per fissare l'angolo verticale (a testa in giù) del pendolo. Terminato il setup, il sensore viene letto a ogni iterazione del loop principale. Un filtro passa basso è applicato alla lettura dell'angolo per smussare il rumore e aumentare la risoluzione; all'angolo viene poi sommato un offset calcolato in modo da essere proporzionale alla lettura del potenziometro presente sulla breadboard, in modo che eventuali

Laboratorio ITSC, report progetto sul pendolo inverso, 28/04/2025
Americo Cherubini is with the Università degli Studi dell'Aquila, L'Aquila, Italy (e-mail: americo.cherubini@student.univaq.it).

imprecisioni di calibrazione possano venire corrette manualmente durante l'operazione del pendolo. L' angolo processato è poi sottoposto a un algoritmo di differenziazione numerica, che ne calcola la derivata rispetto al tempo. I due valori vengono impacchettati in un oggetto e spediti tramite ESP-NOW al MCU di controllo, con una frequenza fissa prestabilita (generalmente poco superiore alla frequenza del loop di controllo). *Fig. 12*

Sul ESP8266 di controllo, la ricezione dei pacchetti ESP-NOW e il relativo aggiornamento delle variabili di stato del pendolo avviene in maniera asincrona: un interrupt è sollevato all'arrivo dei nuovi dati che blocca temporaneamente l'esecuzione del loop principale; i dati ricevuti vengono copiati in uno struct globale, che viene poi letto dal loop di controllo. *Fig. 9*

All'interno del `loop()` è presente la porzione di codice che esegue l'algoritmo di controllo. Questa viene eseguita a una frequenza controllata fissa di 100Hz, e si occupa di calcolare, a partire dallo stato del pendolo attuale, l'ingresso (accelerazione) da applicare al pendolo. L'ingresso viene immagazzinato in una variabile globale `u`, che dunque rimane accessibile fuori da questa porzione di codice, anche per le successive iterazioni del `loop()`. Mentre l'angolo del pendolo è ricevuto come dato remoto, la posizione del carrello (posizione del rotore del motore) viene facilmente calcolata a bordo conoscendo il numero di passi compiuti dal motore (in ipotesi di non perdita di passi). *Fig. 8*

Alla fine del `loop()` è presente la porzione di codice che si occupa del vero e proprio pilotaggio del motore passo passo, che viene eseguita ad ogni iterazione al fine di garantire la maggiore fluidità di rotazione possibile. La classe `PendulumCart` astrae tutta l'interazione con il driver in due principali metodi: `driveAccel()` e `driveSpeed()`. I parametri di accelerazione e velocità sono espressi in cm/s, e si riferiscono alla velocità/accelerazione tangenziale della base del pendolo (così come se fosse ancorato ad un carrello lineare). Il metodo effettivamente utilizzato dagli algoritmi di stabilizzazione è `driveAccel()`; `driveSpeed()` è principalmente utilizzato per motivi di debugging. *Fig. 11*

La generazione degli impulsi di step è realizzata utilizzando la libreria `accellStepper`. L'omonima classe fornita dalla libreria mette a disposizione il metodo `runSpeed()`, che consente di pilotare un motore passo passo alla velocità desiderata. Il principio di funzionamento di `accellStepper` è semplice: il metodo `runSpeed()` deve essere chiamato il più rapidamente possibile all'interno del loop del microcontrollore; se e solo se è ora di effettuare un passo, in base alla velocità selezionata, il metodo genera un brevissimo impulso sul piedino connesso al canale STEP del driver, e regola il piedino connesso a DIR in base alla direzione di rotazione del motore (oraria o antioraria).

I piedini dell'ESP8266 sono collegati direttamente ai canali MS del driver, consentendo il selezionamento del livello di microstepping tramite codice. La classe astratta `MicrostepController` astrae la logica di accensione/spegnimento dei rispettivi piedini MS in un solo metodo: `setMicrostep()`. Una sottoclasse chiamata `A4988microstepController` implementa la specifica logica richiesta dall'omonimo driver per selezionare i vari livelli di microstepping; il suo metodo di `setMicrostep()` accetta in ingresso 5 livelli di microstepping *Tab. I.*

Un oggetto di qualunque classe derivata da `MicrostepDriver` è collegabile a un'istanza di `PendulumCart`, permettendo al carrello di scegliere in maniera adattiva il livello di microstepping, garantendo la migliore performance per la velocità di rotazione richiesta.

A. Logica di selezionamento dinamico del microstepping

Il livello di microstepping viene scelto in maniera automatica in funzione della velocità angolare desiderata, affinché l'operazione del motore sia fluida a basse velocità ma reattiva qualora venissero richiesti movimenti rapidi. L'algoritmo mira a selezionare sempre il più alto livello di microstepping che mantenga la frequenza dei passi sotto una certa soglia massima (la massima frequenza a cui il treno di impulsi può essere generato dal MCU). Assumendo che la dimensione in gradi di un passo sia definita dalla formula $\theta_0 \cdot 2^{-M}$, dove θ_0 è l'angolo di step in assenza di microstepping (1.8deg), e M è il livello di microstepping (con $M \in \{0, 1, 2, 3, 4\}$), si ha che la frequenza di step richiesta per mantenere una data velocità angolare ω è di $f_{step} = \frac{w}{\theta_0} \cdot 2^M$. La formula che viene dunque utilizzata per scegliere il massimo valore di M che mantenga f_{step} sotto la soglia f_{max} per una data ω, è:

$$M = \text{clamp}\left(\text{floor}\left(\log_2\left(f_{max} \cdot \frac{\theta_0}{|\omega|}\right)\right),\ 0,\ 4\right)$$

Dove la funzione $\text{clamp}(n, min, max)$ forza il valore n nel range $[min, max]$.

IV. Interazione tramite seriale

Nel loop del controllore, la funzione `processSerialCommands()` viene chiamata ad ogni iterazione e si occupa di ricevere ed eseguire eventuali comandi testuali inviati alla scheda tramite seriale. Ecco alcuni dei comandi più utili per interfacciarsi con il pendolo:

- `set-target` <*target*>: cambia la posizione di riferimento verso cui il pid secondario tenta di pilotare il carrello.
- `gather-data <ON/OFF/ONBIN>`: ad ogni iterazione del loop di controllo, verrà scritta su seriale una riga contenente le variabili di stato del sistema e un timestamp, in formato .csv (o in formato binario nel caso dell'opzione "ONBIN"). Utile per raccogliere i dati e graficare / studiare l'andamento del sistema.
- `get-pid1/get-pid2`: scrive su seriale i gain del pid indicato.
- `set-pid1/set-pid2` <*P*> <*I*> <*D*>: sovrascrive i gain del pid indicato con quelli passati come parametri (non permanente al reboot).

E' possibile interfacciarsi con il pendolo attraverso qualsiasi applicativo software che supporti la comunicazione su una porta seriale. Il modo più semplice è utilizzare l'IDE Arduino, che incorpora nativamente strumenti per l'invio e ricezione di dati tramite le porte seriale su cui sono connessi i microcontrollori.

Alternativamente, è stata realizzata una libreria Python, reperibile su GitHub, per facilitare notevolmente l'interazione con il pendolo tramite uno script; ecco un breve snippet Python di esempio che utilizza la suddetta libreria per raccogliere i dati sullo stato dal pendolo e salvarli in un file .csv:

```
from serial_pendulum import InvertedPendulum

# passa la porta seriale su cui e' collegato
pendulum = InvertedPendulum("COM8")

esito: bool = pendolo.connect()
print("Pendolo connesso: ", esito)

# abilita lettura dello stato
pendolo.gather_data("BIN")

# ripulisce il buffer di ricezione seriale
pendolo.clear_serial()
```

```
# apre/crea un file con nome "stato.csv"
with open("stato.csv", "a") as f:
    while True:
        #legge lo stato
        state = pendolo.read_state()
        if state:
            print(state)
            pendolo.save_state_to_file(state, f)
```

Una lista estesa di tutti i comandi disponibili può essere ottenuta dal microcontrollore stesso, digitando il comando ”help”.

V. Tecniche di controllo in dettaglio

Per stabilizzare il pendolo, è stato scelto di utilizzare due controllori PID, che agiscono parallelamente sull'ingresso u da imprimere al sistema.

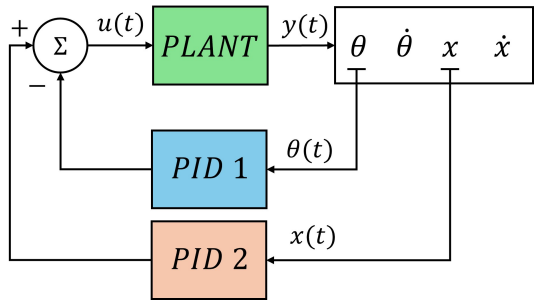


Fig. 3: Diagramma a blocchi del sistema di controllo con due controllori PID.

Il primo PID, a cui ci si riferirà con il nome PID primario, è il PID di stabilizzazione vero e proprio. Esso reagisce direttamente sull'angolo del pendolo e tenta di portarlo a zero. I suoi gain sono calibrati in maniera molto aggressiva in modo da garantire sempre una risposta rapida e decisa. Tuttavia, avere un unico PID di controllo comporta alcuni problemi. In primis, il controllore è molto sensibile a errori di offset costanti della lettura dell'angolo: se la verticale virtuale è spostata rispetto a quella reale, l'unico modo in cui il pendolo può rimanere in “equilibrio” (raggiungere un errore dell'angolo pari a zero) sarebbe imprimendo sempre un'accelerazione costante. Questo, oltre a non essere il goal da raggiungere, è chiaramente irrealizzabile in un modello hardware, a causa dei limiti di velocità del motore. In secondo luogo, anche considerando il caso ideale in cui la verticale software coincida con quella reale, un unico PID non garantisce che, effettuata la stabilizzazione, il carrello abbia velocità nulla. Questo perché l'ingresso al sistema u corrisponde esattamente all'accelerazione del carrello, e poiché la velocità è l'integrale dell'accelerazione nel tempo, NON è garantito il fatto che $\int_0^\infty u\,(t)\,dt \quad = \quad 0$.

E' dunque necessario aggiungere un secondo controllore che agisce sulla posizione del carrello (la prende come input), a cui ci si riferirà con il nome PID secondario. In particolare, questo PID tenta di minimizzare la differenza fra la posizione del carrello e una posizione di target, consentendo il piazzamento del carrello in una posizione arbitraria. Poiché i due controllori PID agiscono in parallelo su due varabili di stato diverse, il tuning dei parametri deve essere effettuato con cautela, onde evitare che i due controllori interferiscano troppo fra di loro. Per evitare che la calibrazione dei PID risulti troppo difficoltosa, si è cercato di separare il più possibile i due controllori in termini di “velocità” (rapidità di convergenza); in questo modo, i controllori lavorano su scale temporali diverse e interferiscono fra di loro in maniera molto più contenuta. Il PID secondario è infatti calibrato in modo da avere una risposta apprezzabilmente più lenta e meno aggressiva rispetto al PID primario; così facendo, i due PID diventano calibrabili in maniera quasi del tutto indipendente fra loro, velocizzando notevolmente il processo.

E' da notare inoltre che, con l'aggiunta del PID secondario, il problema del possibile disallineamento fra la verticale virtuale e reale viene risolto; infatti, qualora ci fosse un disallineamento, l'azione delle componenti P-I del controllore secondario porterebbe comunque il carrello a frenare e stabilizzarsi su una posizione fissa *(L'errore di posizione a regime può essere facilmente calcolato con la formula* $\frac{P_1 \cdot \theta_{err}}{P_2}$*, come si evince dalla Fig. 7)*. Incorporando anche la componente integrale nel controllore, l'errore steady state sulla posizione del carrello che si genera viene gradualmente compensato fino al raggiungimento dello stato desiderato.

VI. Risultati sperimentali

Utilizzando i metodi discussi nella *section IV*, è stato possibile registrare lo stato del pendolo durante varie sessioni e studiarne il comportamento al variare dei parametri dei controllori PID. Nei seguenti grafici, è visualizzato l'andamento dell'angolo e del carrello nel tempo in risposta a una perturbazione impressa dall'esterno.

Nella figura 4 sono stati usati dei gain di `P=3000 I=0 D=200` per il PID primario, e dei gain di `P=0.1 I=0.2 D=5` per il PID secondario. Si noti come l'andamento oscillatorio del pendolo impiega qualche secondo a smorzarsi del tutto, raggiungendo una deviazione massima di angolo di 10 deg.

Nella figura 5 sono stati usati dei gain di `P=10000 I=0 D=550` per il PID primario, e dei gain di `P=3 I=0.3 D=15` per il PID secondario. Si noti come l'oscillazione causata dalla perturbazione si estingua in maniera molto più rapida (circa 1 secondo), e la deviazione di angolo sia più contenuta rispetto a quanto mostrato in *Fig. 4*

Nella figura 6 vengono usati i gain precedenti *(Fig. 5)*, ma il disturbo applicato è notevolmente più accentuato, con una deviazione iniziale dell'angolo di circa 20 deg. Il controllore rapido riesce comunque a governare il sistema e stabilizzarlo nell'arco di circa un secondo.

VII. Conclusioni

E' dunque necessario aggiungere un secondo controllore che agisce sulla posizione del carrello (la prende come input), a cui ci si riferirà con il nome PID secondario. In particolare, questo PID tenta di minimizzare la differenza fra la posizione del carrello e una posizione di target, consentendo il piazzamento del carrello in una posizione arbitraria. Poiché i due controllori PID agiscono in parallelo su due varabili di stato diverse, il tuning dei parametri deve essere effettuato con cautela, onde evitare che i due controllori interferiscano troppo fra di loro. Per evitare che la calibrazione dei PID risulti troppo difficoltosa, si è cercato di separare il più possibile i due controllori in termini di “velocità” (rapidità di convergenza); in questo modo, i controllori lavorano su scale temporali diverse e interferiscono fra di loro in maniera molto più contenuta. Il PID secondario è infatti calibrato in modo da avere una risposta apprezzabilmente più lenta e meno aggressiva rispetto al PID primario; così facendo, i due PID diventano calibrabili in maniera quasi del tutto indipendente fra loro, velocizzando notevolmente il processo.

Appendix

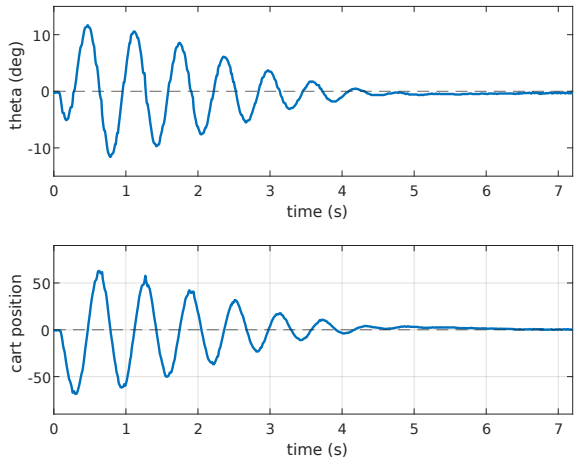


Fig. 4: Risposta a perturbazione con controllore lento.

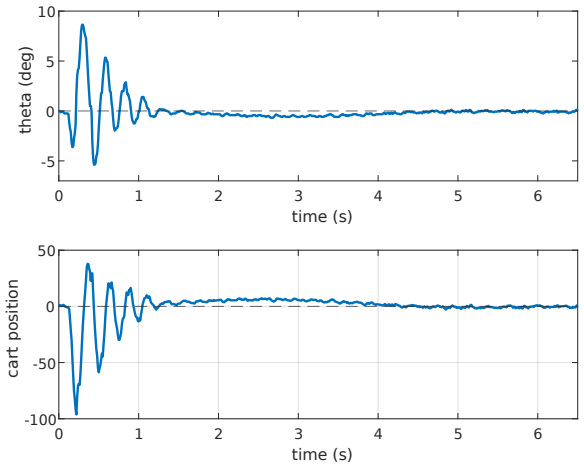


Fig. 5: Risposta a perturbazione con controllore rapido.

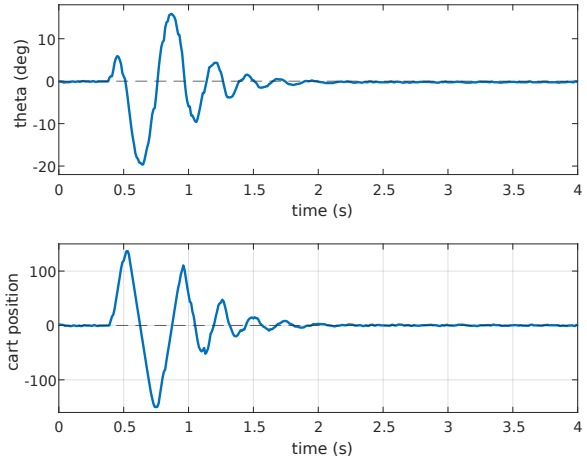
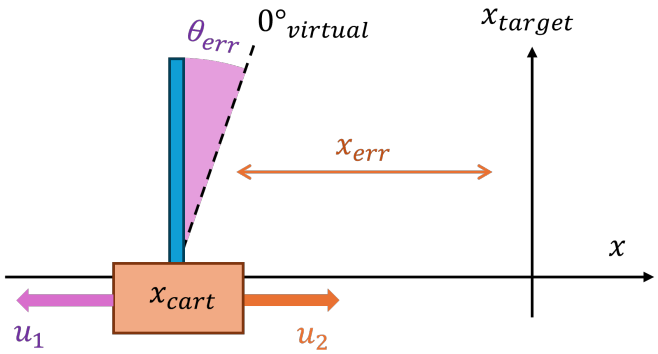


Fig. 6: Risposta a forte perturbazione con controllore rapido.



$$u_1 = P_1 \cdot \theta_{err} \quad = \quad u_2 = P_2 \cdot (x_{target} - x_{cart})$$

Fig. 7: Free Body Diagram dello stato limite del sistema con disallineamento della verticale e in assenza di azioni integrali sull'input.
Se il pendolo è fermo l'ingresso è nullo, dunque le componenti di input dei due PID sono uguali e opposte. Poiché il pendolo è fermo, la componente derivata non compare nell'equazione di equilibrio.

0	No microstepping
1	1/2 microstepping
2	1/4 microstepping
3	1/8 microstepping
4	1/16 microstepping

TABLE I: Livelli di microstepping
La frazione indica il rapporto fra il passo originale di 1.8 deg *e il nuovo passo con microstepping*

```
...
    // depending on
    switch (currentMode) {
    case ControllerType::PID:
      u = loopPID(dt);
      break;
    case ControllerType::FULL_SF:
      u = loopSF(dt);
      break;
    case ControllerType::SPEED:
      u = 0.0;
      break;
    }
    break;
...
```

Fig. 8: Codice di calcolo dell'ingresso u.
Nota: Il codice supporta anche altri tipi di controllori; il meccanismo di calcolo dell'ingresso tuttavia rimane lo stesso.

```
// ESP-NOW on reacv callback.
void OnDataRecv(uint8_t * mac, uint8_t *incomingData, uint8_t len) {
  // copies the whole packet inside angle_packet
  memcpy(&angle_packet, incomingData, sizeof(angle_packet));
}
```

Fig. 9: Funzione per la ricezione dei dati.

```
/* Schema di controllo a due pid a cascata: il pid interno tiene il pendolo dritto,
il pid esterno porta il carrello nella posizione indicata da "target" */
double loopPID(double dt) {
  double r = pid_outer.control(dt, cart_target - state.pos);
  double acc = pid_inner.control(dt, state.theta);

  double u = r - acc;
  return -u; // restituisce l'ingresso
}
```

Fig. 10: Codice di esecuzione dei controllori PID.

```
// aggiorna lo stepper fuori dal loop a 100Hz, per avere una risposta più fluida possibile.
const float dt_real = clock_outer.getdt();
u = constrain(u, -INPUT_RANGE, INPUT_RANGE);

if (currentMode == ControllerType::SPEED) cart.driveSpeed(direct_speed_drive);
else cart.driveAccel(dt_real, u);
```

Fig. 11: Snippet di codice per il pilotaggio del motore stepper (eseguito a ogni iterazione del `loop()`).

```
void loop()
{
    const double dt = loopclock.getdt();

    double new_theta = (AngleSensor.readAngle() * AS5600_RAW_TO_RADIANS_double - angle_offset + angle_pot_offset);
    new_theta = lerp(angle_state.theta, new_theta, 1);
    angle_state.theta_dot = lerp(angle_state.theta_dot, (new_theta - angle_state.theta) / dt, .2);
    angle_state.theta = new_theta;

    // Send message via ESP-NOW, if the time has come
    if (loopclock.tickUs(LOOP_PERIOD)) {
      esp_now_send(receiverAddress, (uint8_t *) &angle_state, sizeof(angle_state));
    }

    // every 200ms update offset from potentiometer
    if (loop_secondary.tickMs(200)) {
      const float pot = multisampleAnalog(A0, 150);
      angle_pot_offset = lerp(angle_pot_offset, (pot - 511) * POT_OFFSET_SCALE, .5);
    }

    delay(2); // 0.002 seconds
}
```

Fig. 12: Codice di loop nel MCU di lettura.