



VAB - Veicolo auto bilanciato

Relazione di progetto

Laboratorio di Sistemi Meccatronici II
Università degli Studi di Bergamo

Kilometro rosso

A.A. 2019/2020

CALEGARI ANDREA - 1041183
PIFFARI MICHELE - 1040658

September 14, 2020

Contents

1	Introduzione	1
2	Dinamica	3
2.1	Scomposizione del VAB	3
2.1.1	Considerazioni iniziali	3
2.1.2	Grandezze di supporto	4
2.1.3	Calcolo componenti dinamiche e potenziali per ogni corpo rigido del sistema . .	4
2.2	Equazioni del moto	9
2.2.1	Lagrangiana	9
2.2.2	Rapporto di trasmissione	10
2.2.3	Sistema non lineare	10
2.2.4	Linearizzazione	11
2.3	Motore	14
3	Controllo	19
3.1	Introduzione	19
3.2	Analisi in anello aperto	19
3.3	Controllore	20
3.3.1	Posizionamento dei poli continui	21
3.3.2	Setpoint di velocità	23
3.4	OPC - UA	24
3.4.1	Idea base OPC-UA	24
3.5	OPC-UA e V.A.B.	25
3.6	RTOS - XENOMAI	26
4	Simulink	29
4.1	Introduzione	29
4.2	Simulazione del sistema non lineare	30
4.3	Simulazione del sistema lineare	31
4.4	Modello complessivo (al momento)	34
4.5	Discretizzazione	34
4.6	Sensori	36
A		41
B		45

List of Figures

2.1	Baricentri dei singoli corpi rigidi	4
2.2	Grandezze di supporto	5
2.3	Ragionamento per il calcolo dei contributi dinamici dello chassis	8
2.4	Equazioni non lineari in forma matriciale	10
2.5	Modellizzazione della parte elettrica del motore	14
2.6	Diagrammi di Bode modello del motore in anello aperto	15
2.7	Schema a blocchi della modellizzazione del motore	16
2.8	Dettaglio del modello del motore	16
3.1	Luogo delle radici del V.A.B. modellizzato in anello aperto	20
3.2	Risposta del sistema non lineare in anello aperto	20
3.3	Schema concettuale del controllo ([1])	21
3.4	Root locus del sistema in anello chiuso	22
3.5	chiusura dell'anello di controllo di ϕ con un controllore I [1]	23
3.6	Taratura del controllore	23
3.7	Schema di massima dell'utilizzo di Raspberry Pi 3	24
3.8	Parametri impostabili lato client	26
3.9	Per specificare quale file .c compilare	27
3.10	Definizione e temporizzazione del ciclo <i>WHILE</i>	27
3.11	Esecuzione a tempo fissato (1 ms) delle funzioni di lettura e calcolo	28
4.1	Risposta del sistema non lineare in anello chiuso	29
4.2	Risposta del sistema lineare in anello chiuso	30
4.3	Implementazione simulink delle equazioni differenziali	30
4.4	Risposta in anello aperto del sistema reale	31
4.5	Implementazione simulink del sistema linearizzato	31
4.6	Risposta in anello aperto del sistema lineare	32
4.7	Transitorio del motore	32
4.8	Zoom del grafico in figura Fig.4.7	33
4.9	Clamp della coppia erogata da parte del motore	33
4.10	Simulink	34
4.11	Modello Simulink del controllore discreto (vista ad alto livello e vista interna)	35
4.12	Modello Simulink del motore discreto	36
4.13	Blocco Simulink responsabile della generazione del rumore	36
4.14	Modello Simulink dei sensori <i>encoder</i> e <i>I.M.U.</i> con Moving Average	37
4.15	Modello Simulink dei sensori <i>encoder</i> e <i>I.M.U.</i> con Filtro di Kalman	38
A.1	Diagramma di flusso rappresentante le funzionalità del server	41

1

Introduzione

L'approccio seguito per la stesura del modello dinamico del veicolo autobilanciato ha da subito preso una via meno *tradizionale* rispetto al classico metodo risolutivo: abbiamo infatti preferito, dato il nostro *background* informatico, approcciare il problema direttamente in ambiente Matlab, sfruttando sin da subito le potenzialità di calcolo offerte dal software di *Mathworks*.

Nello specifico, per la parte di stesura e definizione della dinamica, abbiamo inizialmente seguito una via risolutiva duale, portando avanti sia un'analisi letterale, sfruttando le potenzialità del **calcolo simbolico** messe a disposizione delle funzionalità di **live scripting**, sia uno studio numerico (considerando quindi le varie grandezze fisiche con i valori definiti delle specifiche di progetto).

In linea di massima lo sviluppo del progetto ha seguito un andamento a step gradualmente, cadenzati da incontri settimanali in cui poter confrontare e consolidare lo *stato di avanzamento dei lavori*: nello specifico, il lavoro ha seguito uno sviluppo in questa direzione step by step, rappresentabile in linea di massima da queste *pietre miliari*:

- **Dinamica di ogni singolo corpo rigido**: abbiamo impostato il problema della dinamica andando a considerare il veicolo auto bilanciato come un insieme di corpi rigidi di cui poterne studiare la dinamica in maniera separata;
- **Dinamica completa del VAB**: siamo andati poi a considerare il sistema nella sua completezza, unendo i contributi dei corpi rigidi considerati in prima battuta singolarmente. Questo ci ha permesso di ottenere le equazioni del moto, in forma non lineare, le quali hanno permesso una completa descrizione del sistema che abbiamo modellizzato;
- **Linearizzazione**: siamo poi andati a linearizzare queste equazioni dinamiche (appunto non lineari) nell'intorno dell'equilibrio;
- **Definizione del controllo**: tramite la tecnica di *pole placement*, siamo andati a definire il controllore più adatto per questo sistema;
- **Modellizzazione del motore**: introduzione del modello del motore sulla base delle caratteristiche reali contenute nella specifica di progetto;
- **Discretizzazione**: passaggio a tempo discreto per i segnali derivanti dal mondo analogico, ovvero per tutto ciò che concerne la parte di sensoristica;
- **Non idealità**: siamo andati a modellizzare anche la presenza di disturbi, di natura stocastica e legati alla quantizzazione;
- **Trasformazioni di blocchi in *interpreted function***: conversione delle funzionalità di controllo del motore e filtraggio dei segnali provenienti dai sensori in blocchi di codice *Matlab* per favorirne poi la successiva conversione ed implementazione a bordo del controllore;

2

Dinamica

2.1 Scomposizione del VAB

2.1.1 Considerazioni iniziali

Per il calcolo delle equazioni dinamiche del sistema siamo andati a considerare ogni singolo corpo rigido componente il sistema, calcolandone le grandezze fisiche di posizione e velocità, seguendo un approccio cartesiano. Nello specifico abbiamo considerato il sistema composto da:

- Asta
- Utente a bordo dello chassis
- Chassis (nel corso della trattazione sarà chiamata talvolta anche base)
- Ruota (che poi sarà considerata con un contributo doppio, essendo il VAB composto da due ruote)

Ognuno di questi corpi rigidi separati è individuato da un punto, che ne rappresenta il centro di massa (o baricentro del corpo stesso): avremo quindi il seguente insieme di punti caratterizzanti il sistema (figura 2.1)

- P_a
- P_b
- P_c
- P_r

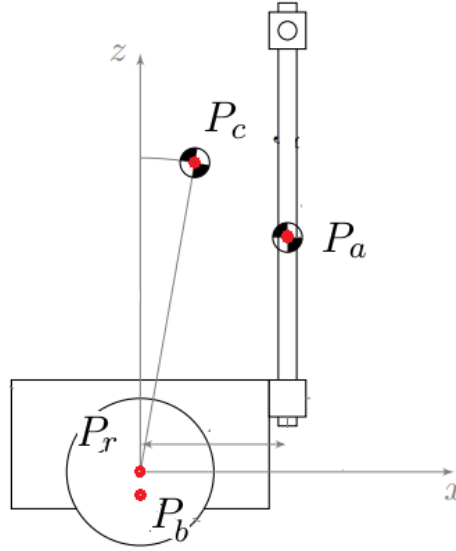


Figure 2.1: Baricentri dei singoli corpi rigidi

2.1.2 Grandezze di supporto

Prima di andare a definire le componenti di energia potenziale e cinetica di ogni singolo corpo, siamo andati ad introdurre alcune grandezze geometriche di supporto che definiremo qui di seguito.

Nello specifico abbiamo introdotto i seguenti parametri, specificati anche in figura 2.2:

- l_a : rappresenta la congiungente tra il centro del sistema di riferimento XZ e il centro dell'asta, utilizzata appunto come manubrio, individuato come

$$l_a = \sqrt{\left(\frac{h_a}{2} + \frac{h_b}{2}\right)^2 + \left(\frac{w_b}{2}\right)^2}$$

- l_c : questa grandezza rappresenta per noi l'altezza del baricentro del corpo dell'utente, la quale ovviamente andrà a dipendere dal valore di inclinazione del corpo stesso. Considerando il corpo inizialmente in posizione verticale, avremo che questa lunghezza corrisponde alla congiungente dal centro del sistema di riferimento al punto P_c , che equivale a dire:

$$l_c = 0.55h_c + \frac{h_b}{2}$$

- l_b : rappresenta lo spostamento verso il basso, lungo l'asse z, del baricentro dello chassis. Da specifiche del progetto sappiamo che questa grandezza ha valore (con segno negativo) di

$$l_b = 0.1m$$

- β : angolo formato con la verticale dalla congiungente tra il centro del sistema di riferimento e il punto P_a . Si ricava, con un semplice approccio trigonometrico che, l'angolo in questione, ha questa forma:

$$\arctan\left(\frac{\frac{w_b}{2}}{\frac{h_a}{2} + \frac{h_b}{2}}\right)$$

2.1.3 Calcolo componenti dinamiche e potenziali per ogni corpo rigido del sistema

Per ognuno dei corpi rigidi definiti in precedenza siamo andati appunto a calcolare:

- **Coordinate spaziali \mathbf{P}** espresse nel sistema di riferimento XZ. A queste due coordinate cartesiane ne va aggiunta una terza, relativa alle coordinate angolari (per poter tener così conto dei contributi inerziali dei corpi);

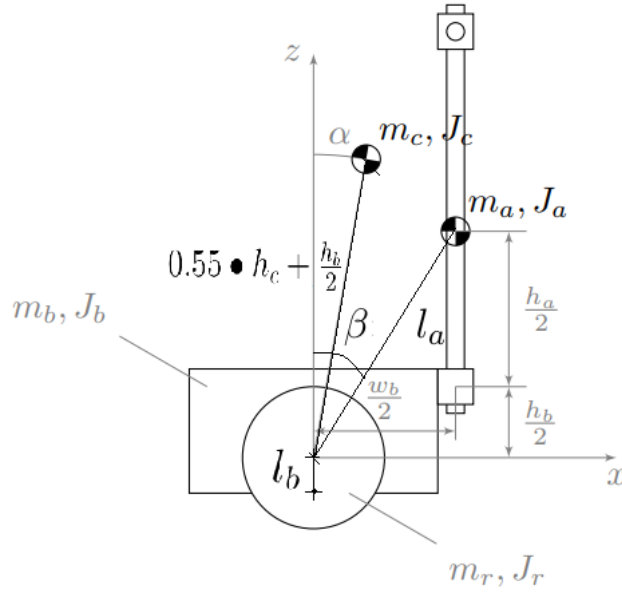


Figure 2.2: Grandezze di supporto

- **Vettore delle velocità** $\mathbf{V} \rightarrow$ vettore 3×1
- **Matrice delle masse** $\mathbf{M} \rightarrow$ matrice 3×3
- **Energia cinetica** $\mathbf{T} \rightarrow \frac{1}{2} \mathbf{V}^T \mathbf{M} \mathbf{V}$
- **Energia potenziale** \mathbf{U}
- **Lagrangiana parziale** \mathbf{L} del singolo corpo rigido

Asta

- $$P_a = \begin{pmatrix} r \phi(t) + l_a \sin(\beta + \theta(t)) \\ l_a \cos(\beta + \theta(t)) \\ \theta(t) \end{pmatrix}$$
- $$V_a = \begin{pmatrix} r \dot{\phi}(t) + l_a \cos(\beta + \theta(t)) \dot{\theta}(t) \\ -l_a \sin(\beta + \theta(t)) \dot{\theta}(t) \\ \dot{\theta}(t) \end{pmatrix}$$
- $$M_a = \begin{pmatrix} m_a & 0 & 0 \\ 0 & m_a & 0 \\ 0 & 0 & m_a l_a^2 + J_a \end{pmatrix}$$
- $$T_a = m_a l_a^2 (\dot{\theta}(t))^2 + \frac{m_a r^2 (\dot{\phi}(t))^2}{2} + \frac{J_a (\dot{\theta}(t))^2}{2} + m_a \cos(\beta + \theta(t)) l_a r \dot{\theta}(t) \dot{\phi}(t)$$
- $$U_a = g l_a m_a \cos(\beta + \theta(t))$$
- $$L_a = m_a l_a^2 (\dot{\theta}(t))^2 + \frac{m_a r^2 (\dot{\phi}(t))^2}{2} + \frac{J_a (\dot{\theta}(t))^2}{2} + m_a \cos(\beta + \theta(t)) l_a r \dot{\theta}(t) \dot{\phi}(t) - g m_a \cos(\beta + \theta(t)) l_a$$

Chassis (base)

- $P_b = \begin{pmatrix} r \phi(t) - l_b \sin(\theta(t)) \\ -l_b \cos(\theta(t)) \\ \theta(t) \end{pmatrix}$
- $V_b = \begin{pmatrix} r \dot{\phi}(t) - l_b \cos(\theta(t)) \dot{\theta}(t) \\ l_b \sin(\theta(t)) \dot{\theta}(t) \\ \dot{\theta}(t) \end{pmatrix}$
- $M_b = \begin{pmatrix} m_b & 0 & 0 \\ 0 & m_b & 0 \\ 0 & 0 & m_b l_b^2 + J_b \end{pmatrix}$
- $T_b = m_b l_b^2 (\dot{\theta})^2 + \frac{m_b r^2 (\dot{\phi})^2}{2} + \frac{J_b (\dot{\theta})^2}{2} - m_b \cos(\theta(t)) l_b r \dot{\theta} \dot{\phi}$
- $U_b = -g l_b m_b \cos(\theta(t))$
- $L_b = m_b l_b^2 (\dot{\theta})^2 + \frac{m_b r^2 (\dot{\phi})^2}{2} + \frac{J_b (\dot{\theta})^2}{2} - m_b \cos(\theta(t)) l_b r \dot{\theta} \dot{\phi} + g m_b \cos(\theta(t)) l_b$

Utente

- $P_c = \begin{pmatrix} r \phi(t) + l_c \sin(\alpha + \theta(t)) \\ l_c \cos(\alpha + \theta(t)) \\ \alpha + \theta(t) \end{pmatrix}$
- $V_c = \begin{pmatrix} r \dot{\phi}(t) + l_c \cos(\alpha + \theta(t)) \dot{\theta}(t) \\ -l_c \sin(\alpha + \theta(t)) \dot{\theta}(t) \\ \dot{\theta}(t) \end{pmatrix}$
- $M_c = \begin{pmatrix} m_c & 0 & 0 \\ 0 & m_c & 0 \\ 0 & 0 & m_c l_c^2 + J_c \end{pmatrix}$
- $T_c = m_c l_c^2 (\dot{\theta})^2 + \frac{m_c r^2 (\dot{\phi})^2}{2} + \frac{J_c (\dot{\theta})^2}{2} + m_c \cos(\alpha + \theta(t)) l_c r \dot{\theta} \dot{\phi}$
- $U_c = g l_c m_c \cos(\alpha + \theta(t))$
- $L_c = m_c l_c^2 (\dot{\theta})^2 + \frac{m_c r^2 (\dot{\phi})^2}{2} + \frac{J_c (\dot{\theta})^2}{2} + m_c \cos(\alpha + \theta(t)) l_c r \dot{\theta} \dot{\phi} - g m_c \cos(\alpha + \theta(t)) l_c$

Ruota

$$\bullet P_r = \begin{pmatrix} r \phi(t) \\ 0 \\ \phi(t) \end{pmatrix}$$

$$\bullet V_r = \begin{pmatrix} r \dot{\phi}(t) \\ 0 \\ \dot{\phi}(t) \end{pmatrix}$$

$$\bullet M_r = \begin{pmatrix} m_r & 0 & 0 \\ 0 & m_r & 0 \\ 0 & 0 & J_r \end{pmatrix}$$

$$\bullet T_r = \frac{(m_r r^2 + J_r) (\dot{\phi}(t))^2}{2}$$

$$\bullet U_r = 0$$

$$\bullet L_r = \frac{(m_r r^2 + J_r) (\dot{\phi}(t))^2}{2}$$

Veicolo completo

Una volta trovati le componenti dinamiche dei singoli corpi rigidi, possiamo andare a definire l'energia cinetica e potenziale totale del sistema, per poter poi calcolare l'equazione di Lagrange per l'intero sistema. In sostanza quindi avremo:

$$L = L_a + L_b + L_c + 2L_r$$

Quello che ne deriva è quindi la seguente equazione Lagrangiana, in grado di descrivere la dinamica dell'intero sistema (riportiamo il risultato letterale, ovvero non legato alla sostituzione di alcun valore numerico che descrive il sistema):

$$L = \left(m_a l_a^2 + m_b l_b^2 + m_c l_c^2 + \frac{J_a}{2} + \frac{J_b}{2} + \frac{J_c}{2} \right) (\dot{\theta}(t))^2 + \left(m_r r^2 + J_r + \frac{m_a r^2}{2} + \frac{m_b r^2}{2} + \frac{m_c r^2}{2} \right) (\dot{\phi}(t))^2 +$$

$$\left(m_c \sigma_4 l_c r + m_a \sigma_3 l_a r - m_b \cos(\theta(t)) l_b r \right) \dot{\theta}(t) \dot{\phi}(t) + g m_b \cos(\theta(t)) l_b - g m_c \sigma_4 l_c - g m_a \sigma_3 l_a$$

$$\sigma_3 = \cos(\beta + \theta(t)) \sigma_4 = \cos(\alpha + \theta(t))$$

Note sul calcolo delle componenti dinamiche

- Nel calcolo della matrice di massa abbiamo considerato tre componenti:
 - Componente di massa lungo x;
 - Componente di massa lungo z;
 - Componente di massa rotazionale: dalla meccanica è noto che un corpo, con una certa massa che si trova in uno stato di rotazione, avrà un contributo inerziale che dipende dal braccio rispetto all'asse intorno al quale avviene la rotazione.
- Nello specifico abbiamo considerato il teorema di *Huygens-Steiner* per ogni singolo corpo rigido che è stato preso in esame.

$$I_a = I_{cm} + m d^2$$

Esso permette di definire l'inerzia di un corpo come la somma di due diverse componenti:

- * Momento d'inerzia definito rispetto all'asse passante per il centro di massa: questo parametro rappresenta il valore che è fornito dalle specifiche del progetto per gli specifici corpi;
- * Prodotto tra la massa m del corpo preso in considerazione e la distanza tra l'asse rispetto a cui si riferisce la rotazione e quello passante per il centro di massa;

Questa componente inerziale è visibile nelle matrici di massa in posizione (3,1): si sottolinea come invece, per la matrice di massa relativa alle ruote (matrice M_r 2.1.3), non sia presente la componente esplicitata dal teorema di *Huygens-Steiner* per il fatto che il centro di massa della ruota coincide con quello del sistema di riferimento XZ e quindi di conseguenza non è necessaria alcuna traslazione di asse;

- Nel calcolo simbolico in ambiente *Matlab* siamo andati ad utilizzare le funzionalità di *collect* e *simplify*, per permettere di ridurre e semplificare le equazioni. Nello specifico, con il comando *simplify*, tramite l'opzione *Steps*, siamo andati a settare il numero di step che l'algoritmo di calcolo simbolico andrà ad eseguire per poter ridurre e semplificare il maggior numero di termini possibili (\uparrow *Steps*, \uparrow *Compattezza eq symb*);
- Le ruote, nel calcolo della dinamica completa del VAB, sono state considerate con un contributo doppio;
- Nello studio dello chassis (base), abbiamo seguito questo approccio: il baricentro della base stessa sappiamo essere posizionato ad una quota differente rispetto al centro del sistem di coordinate XZ preso come riferimento. Per questo motivo il suo contributo in termini cinetici e potenziali dipende dal valore dell'inclinazione dello chassis, ovvero dal valore dell'angolo *caratterizzante* il sistema θ : questo concetto è evidenziato in figura 2.3. L'aggiunta di π al valore di θ è necessaria per poter rendere sensibile i valori di energia cinetica e potenziale al *quadrante* in cui si trova ad essere posizionato il centro di massa della base stessa (P_b).

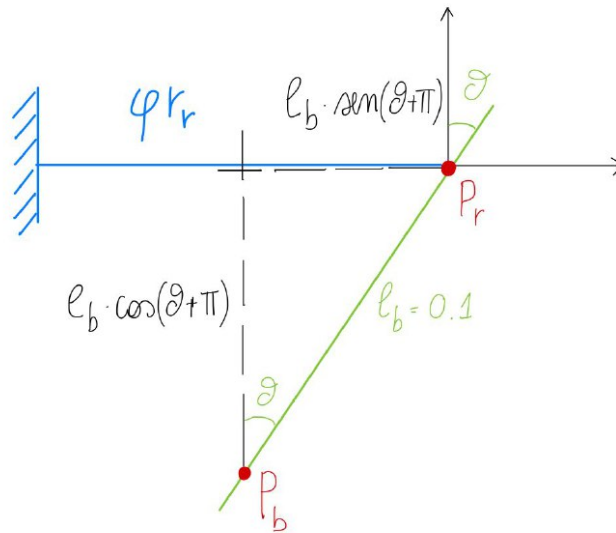


Figure 2.3: Ragionamento per il calcolo dei contributi dinamici dello chassis

2.2 Equazioni del moto

2.2.1 Lagrangiana

Una volta definita la dinamica del sistema ed ottenuta quindi l'equazione di Lagrange che ne caratterizza il comportamento, andiamo a ricavare le equazioni del moto, le quali consentono di definire l'andamento delle *coordinate libere* (scelte in fase iniziale di progetto) che sappiamo essere

- $\phi = q_1 = \text{angolo di rotazione delle ruote}$
- $\theta = q_2 = \text{angolo di inclinazione dello chassis}$

In particolare possiamo definire le cosiddette *equazioni di Eulero-Lagrange*, ovvero un insieme di n equazioni differenziali (con n pari al numero di coordinate libere del sistema), la cui risoluzione fornisce le equazioni del moto del sistema.

$$\frac{d}{dt} \left(\frac{\partial}{\partial \dot{q}_i} L \right) - \frac{\partial}{\partial q_i} L = Q_i \rightarrow i = 1, \dots, n$$

Sono da effettuare alcune annotazioni sugli addenti presenti nell'equazione precedente (eq 2.2.1):

- q rappresenta la coordinata libera (nel nostro caso saranno θ e ϕ);
- al secondo membro della lagrangiana troviamo le componenti generalizzate, relative alle forze attive esterne. Nel nostro sistema andiamo a modellizzare, come forze attive, quelle che sono le forze motrici immesse dai motori che gestiscono il movimento del V.A.B.: in particolare, nel nostro caso, per il legame matematico tra le componenti generalizzate e il concetto di lavoro virtuale, avremo che l'equazione lagrangiana sarà equiparata al valore di torque immessa dal motore nel sistema;
- molto importante, ai fini della correttezza delle equazioni del moto, è il valore da assegnare al secondo membro dell'equazione (altro non è che la componente di *torque*, ovvero la forza esterna che agisce); infatti esso varia a seconda che il sistema agisca sulla coordinata libera q a valle o a monte della trasmissione.

Nello specifico, si osserva che il motore è composto da due parti: lo statore e il rotore. Esse trasmettono una coppia uguale e inversa ai corpi a cui sono solidalmente collegati: lo statore, posizionato nella base del Segway, influenza la coordinata libera θ ed esercita una coppia sullo chassis pari ad una generica coppia $-2C_m$. Il fattore moltiplicativo 2 è dovuto al fatto che nella base sono presenti due motori.

Nello stesso modo, il rotore, trasmette all'albero motore una coppia C_m : a valle della trasmissione si misura dunque una coppia $\frac{C_m}{\tau}$ dovuta alla presenza della trasmissione. Questa coppia ridotta all'albero dell'utilizzatore è la coppia che in definitiva va ad agire sulle ruote e quindi sulla coordinata ϕ .

Si ottiene dunque il seguente sistema di equazioni, la cui risoluzione permette di ottenere due equazioni differenziali del secondo ordine nelle variabili θ e ϕ :

$$\begin{cases} \frac{d}{dt} \left(\frac{\partial}{\partial \dot{\phi}} L \right) - \frac{\partial}{\partial \phi} L = \frac{C_m}{\tau} \\ \frac{d}{dt} \left(\frac{\partial}{\partial \dot{\theta}} L \right) - \frac{\partial}{\partial \theta} L = -2C_m \end{cases}$$

dove τ è il rapporto di trasmissione il cui calcolo è riportato nella sezione 2.2.2.

Lo stesso sistema di equazioni, che per semplicità non è qui riportato, può essere rappresentato anche in forma matriciale, come mostrato in figura 2.4:

$$\begin{bmatrix} M_1 & M_2 \\ M_3 & M_4 \end{bmatrix} \begin{bmatrix} \ddot{\theta} \\ \ddot{\phi} \end{bmatrix} = - \begin{bmatrix} -l_a m_a r \sin(\beta + \theta(t)) \dot{\theta}^2 + l_b m_b r \sin(\theta(t)) \dot{\theta}^2 - l_c m_c r \sin(\alpha + \theta(t)) \dot{\theta}^2 \\ -g(l_c m_c \cos(\alpha + \theta(t)) + l_a m_a \sin(\beta + \theta(t)) - l_b m_b \sin(\theta(t))) \dot{\theta} - l_a m_a r \sin(\beta + \theta(t)) \dot{\theta} \dot{\phi} + l_b m_b r \sin(\theta(t)) \dot{\theta} \dot{\phi} - l_c m_c r \sin(\alpha + \theta(t)) \dot{\theta} \dot{\phi} \end{bmatrix} + \begin{bmatrix} \frac{C_m}{\tau} \\ -2C_m \end{bmatrix}$$

$$M_1 = l_a m_a r \cos(\beta + \theta(t)) - l_b m_b r \cos(\theta(t)) + l_c m_c r \cos(\alpha + \theta(t))$$

$$M_2 = 2m_r r^2 + 2J_r + m_a r^2 + m_b r^2 + m_c r^2$$

$$M_3 = J_a + J_b + J_c + 2l_a^2 m_a + 2l_b^2 m_b + 2l_c^2 m_c$$

$$M_4 = l_a m_a r \cos(\beta + \theta(t)) - l_b m_b r \cos(\theta(t)) + l_c m_c r \cos(\alpha + \theta(t))$$

Figure 2.4: Equazioni non lineari in forma matriciale

La soluzione di questo sistema di equazioni verrà ad essere utilizzata per rappresentare il comportamento del sistema reale, basandosi su una coppia di equazioni non lineari: questa tematica verrà poi approfondita nella sezione 4.2.

2.2.2 Rapporto di trasmissione

Come da specifiche del progetto, ognuno dei due motori è collegato alle ruote mediante una doppia sequenza di riduttori:

- Un primo riduttore epicicloidale con rapporto di trasmissione $\tau_1 = 0.1$;
- Un secondo riduttore, rappresentato da una cinghia dentata, con un rapporto di trasmissione facilmente calcolabile come rapporto tra il numero dei denti dei due alberi, essendo il passo uguale in entrambi gli alberi.

$$\tau_2 = \frac{\text{Numero_denti_puleggia_ingresso}}{\text{Numero_denti_puleggia_uscita}} = \frac{Z_{in}}{Z_{out}} = \frac{22}{26}$$

Il rapporto di trasmissione completo τ è quindi definito come:

$$\tau = \tau_1 \cdot \tau_2 = 0.085$$

2.2.3 Sistema non lineare

La risoluzione del sistema di equazioni sopra riportato permette di ottenere due equazioni differenziali del secondo ordine: queste soluzioni sono state ottenute sfruttando la potenzialità del calcolo simbolico messo a disposizione da Matlab, come si vede nel listato seguente.

```
1 theta2_diff = subs(theta2,{phi,vel,theta,vel_ang,C_m},{q_1 q_1_p q_2 q_2_p u})
2 phi2_diff = subs(phi2,{phi,vel,theta,vel_ang,C_m},{q_1 q_1_p q_2 q_2_p u })
```

Listing 2.1: Risoluzione del sistema di equazioni

Le soluzioni così ottenute rappresentano e governano la dinamica del sistema: esse verranno quindi utilizzate per andare ad analizzare il comportamento del sistema considerato reale, utilizzando il controllore che è stato progettato sul sistema linearizzato.

Un passaggio importante per la simulazione del sistema non lineare, è quello rappresentato dal salvataggio di queste due equazioni in esame.

Esse sono, per forza di cose, richiamate all'interno della simulazione *Simulink*: per fare ciò abbiamo utilizzato il comando `matlabFunction()`, il quale permette di scrivere in una funzione di matlab le due equazioni in esame.

```
matlabFunction(theta2_diff, 'File', 'theta_secondo');
matlabFunction(phi2_diff, 'File', 'phi_secondo');
```

Listing 2.2: Salvataggio in funzioni Matlab

Nello specifico queste funzioni avranno come input i valori da cui dipendono le equazioni differenziali e, componendoli algebricamente tra di loro, fornirà come output i valori di $\ddot{\theta}$ e $\ddot{\phi}$.

2.2.4 Linearizzazione

Nell'approccio alla modellistica dei sistemi dinamici, uno step molto importante è quello che concerne la linearizzazione del sistema, ovvero il passaggio da un insieme di equazioni non lineari ad un set di equazioni lineari definite nell'intorno di un punto specifico dello stato del sistema: questo nuovo sistema di equazioni andrà a definire un sistema lineare in grado di approssimare il comportamento dinamico del sistema non lineare vicino all'equilibrio.

Il punto d'equilibrio in questione è quello in cui la variazione dello stato del sistema è nullo: detto quindi $\mathbf{x}(t)$ il vettore di stato, l'equilibrio sarà dato da $\mathbf{x}(t)$.

Perciò, il passo successivo alla risoluzione del sistema di equazioni di Lagrange in θ e ϕ che è stato svolto riguarda la linearizzazione: nel contesto del controllo, linearizzare è importante poiché permette di progettare il controllore stesso sul sistema tangente a quello reale (ovvero il sistema lineare), potendo poi testarlo direttamente sul sistema reale (ovvero quello descritto dalle equazioni non lineari).

Calcolo della posizione di equilibrio

Per poter quindi procedere con la linearizzazione è necessario innanzitutto calcolare l'equilibrio del sistema, cioè il punto in cui $\ddot{\theta} = \mathbf{0}$: essa corrisponde ad una situazione in cui lo stato del sistema risulta essere in una condizione di equilibrio dinamico, ovvero quando la sua variazione nel tempo è nulla (nessuna accelerazione \rightarrow velocità costante \rightarrow posizione lineare); tale equilibrio è data dalle due posizioni seguenti:

$$\begin{pmatrix} -0.01174364 - 7.105427e-15i \\ 3.129849 - 7.105427e-15i \end{pmatrix}$$

Queste posizioni di equilibrio le abbiamo ottenute per mezzo della istruzione Matlab presentata di seguito, nella quale si vede come, partendo dall'equazione differenziale del sistema non lineare espressa in termini di $\ddot{\theta}$, si ricava la posizione di equilibrio del sistema.

```
equilibri = solve(subs(theta2_differenziabile, {q_1 q_1_p q_2_p u}, {0,0,0,0})==0, q_2)
```

Listing 2.3: Calcolo posizione di equilibrio

Come si vede, siamo interessati solamente al valore di equilibrio relativo di θ : per questo motivo che poniamo a 0 (valore arbitrario) i valori di ϕ e $\dot{\phi}$, poiché non risultano essere di interesse per il calcolo dell'equilibrio.

I risultati ottenuti sono, con buona approssimazione, considerabili numeri naturali e rispondono a quanto ci aspettavamo: essendo presente un offset (w_b) tra quello che è il baricentro del sistema di riferimento e quello relativo al manubrio, non ci aspettavamo di ottenere due equilibri perfettamente di 0° (posizione verticale a "testa in sù") e di 180° (posizione verticale a "testa in giù"), ma bensì due posizioni leggermente spostate rispetto alla verticale.

Per completezza e per avere comunque un feedback più concreto, abbiamo provato ad azzerare il valore di w_b , andando a ricalcolare la posizione di equilibrio, ottenendo effettivamente i valori di

$$\begin{pmatrix} 0 \text{ deg} \\ 180 \text{ deg} \end{pmatrix}$$

Dal risultato ottenuto, considerando nulla la componente immaginaria, possiamo notare che (osservazioni comuni per i sistemi assimilabili al pendolo):

- il sistema ha due equilibri; il primo ($\theta = -0.01174364 \text{ rad} = -0.67 \text{ deg}$) ci si aspetta che sia instabile in quanto il baricentro del sistema V.A.B. (compreso di utente a bordo) risulta essere sopra all'asse delle ruote;
- il secondo ($\theta = 3.129849 \text{ rad} = 179.32 \text{ deg}$) è un equilibrio stabile, poiché il baricentro sta sotto l'asse delle ruote e, un eventuale disturbo dopo un certo tempo di transitorio, risulterebbe avere effetto nullo sullo stato del sistema, che ritornerebbe alla medesima posizione;

Calcolo del sistema lineare

Ovviamente, ai fini del controllo, si è linearizzato attorno al primo equilibrio; non avrebbe infatti senso controllare il sistema quando questo risulta essere capovolto (cosa che per di più risulta essere fisicamente impossibile, se il veicolo autobilanciato viene fatto muovere su una superficie).

I vettori che definiscono le **variabili di stato del sistema** e le **uscite del sistema** sono i seguenti:

$$\begin{aligned} x = \text{variabili di stato} &= \begin{bmatrix} x_1 \rightarrow \phi \\ x_2 \rightarrow \dot{\phi} \\ x_3 \rightarrow \theta \\ x_4 \rightarrow \dot{\theta} \end{bmatrix} \\ y = \text{output del sistema} &= \begin{bmatrix} y_1 \rightarrow \ddot{\theta} \\ y_2 \rightarrow \ddot{\phi} \end{bmatrix} \end{aligned}$$

Per individuare il risultato finale della linearizzazione, abbiamo seguito un approccio a matrici: nello specifico abbiamo utilizzato la notazione matriciale nel caso di sistemi SIMO (*Single Input Multiple Output*), la cui stesura ha utilizzato le seguenti funzioni di supporto:

$$\left\{ \begin{array}{l} \mathbf{f}_1 \rightarrow \dot{x}_1(t) = \dot{\phi} = x_2(t) = 0 \text{ (poichè all'equilibrio)} \\ \mathbf{f}_2 \rightarrow \dot{x}_2(t) = \ddot{\phi} \\ \mathbf{f}_3 \rightarrow \dot{x}_3(t) = \dot{\theta} = x_4(t) = 0 \text{ (poichè all'equilibrio)} \\ \mathbf{f}_4 \rightarrow \dot{x}_4(t) = \ddot{\theta} \\ \mathbf{g}_1 \rightarrow y_1(t) = \phi = x_1 \\ \mathbf{g}_2 \rightarrow y_2(t) = \dot{\phi} = x_2 \\ \mathbf{g}_3 \rightarrow y_3(t) = \theta = x_3 \\ \mathbf{g}_4 \rightarrow y_4(t) = \dot{\theta} = x_4 \end{array} \right.$$

Sfruttando le caratteristiche *matematiche* del sistema non lineare, ottenibili tramite lo sviluppo in serie di Taylor nell'intorno dell'equilibrio, possiamo definire i termini matriciali che permettono di caratterizzare il sistema, secondo le seguenti equazioni:

$$\begin{cases} \dot{x}(t) = Ax(t) + Bu(t) \\ y(t) = Cx(t) + Du(t) \end{cases}$$

$$\begin{aligned}
A &= \begin{bmatrix} \frac{\partial}{\partial x_1} f_1 & \frac{\partial}{\partial x_2} f_1 & \frac{\partial}{\partial x_3} f_1 & \frac{\partial}{\partial x_4} f_1 \\ \frac{\partial}{\partial x_1} f_2 & \frac{\partial}{\partial x_2} f_2 & \frac{\partial}{\partial x_3} f_2 & \frac{\partial}{\partial x_4} f_2 \\ \frac{\partial}{\partial x_1} f_3 & \frac{\partial}{\partial x_2} f_3 & \frac{\partial}{\partial x_3} f_3 & \frac{\partial}{\partial x_4} f_3 \\ \frac{\partial}{\partial x_1} f_4 & \frac{\partial}{\partial x_2} f_4 & \frac{\partial}{\partial x_3} f_4 & \frac{\partial}{\partial x_4} f_4 \end{bmatrix} & B &= \begin{bmatrix} \frac{\partial}{\partial u} f_1 \\ \frac{\partial}{\partial u} f_2 \\ \frac{\partial}{\partial u} f_3 \\ \frac{\partial}{\partial u} f_4 \end{bmatrix} & C &= \begin{bmatrix} \frac{\partial}{\partial x_1} g_1 & \frac{\partial}{\partial x_2} g_1 & \frac{\partial}{\partial x_3} g_1 & \frac{\partial}{\partial x_4} g_1 \\ \frac{\partial}{\partial x_1} g_2 & \frac{\partial}{\partial x_2} g_2 & \frac{\partial}{\partial x_3} g_2 & \frac{\partial}{\partial x_4} g_2 \\ \frac{\partial}{\partial x_1} g_3 & \frac{\partial}{\partial x_2} g_3 & \frac{\partial}{\partial x_3} g_3 & \frac{\partial}{\partial x_4} g_3 \\ \frac{\partial}{\partial x_1} g_4 & \frac{\partial}{\partial x_2} g_4 & \frac{\partial}{\partial x_3} g_4 & \frac{\partial}{\partial x_4} g_4 \end{bmatrix} \\
D &= \begin{bmatrix} \frac{\partial}{\partial u} g_1 \\ \frac{\partial}{\partial u} g_2 \\ \frac{\partial}{\partial u} g_3 \\ \frac{\partial}{\partial u} g_4 \end{bmatrix}
\end{aligned}$$

Nel specifico del veicolo autobilanciato, nel caso con utente con peso di 70 kg e altezza 1.77 m , siamo andati ad ottenere i seguenti valori numerici per le matrici in esame:

$$\begin{aligned}
A &= \begin{bmatrix} 0 & 1.0 & 0 & 0 \\ 0 & 0 & -15.2286 & 0 \\ 0 & 0 & 0 & 1.0 \\ 0 & 0 & 5.43924 & 0 \end{bmatrix} & B &= \begin{bmatrix} 0 \\ 2.7498 \\ 0 \\ -0.2612 \end{bmatrix} & C &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
D &= \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}
\end{aligned}$$

Si nota come, anche dimensionalmente parlando, le matrici siano corrette con quanto ricavato dalla teoria; definendo n come il numero di gdl e le rispettive derivate (nel nostro caso $n = 4$) abbiamo che:

- $A \rightarrow [2n \times 2n]$
- $B \rightarrow [2n \times 1]$
- $C \rightarrow [2n \times 2n]$: la dimensione in questo caso può essere variabile, a seconda dei parametri che intendiamo riportare verso l'esterno;
- $D \rightarrow [1 \times 2n]$: anche in questo caso la dimensione è collegata alla scelta fatta per la matrice C;

Ricordando infine che un generico sistema dinamico può essere scritto come

$$G(s) = C(sI - A)^{-1}B + D$$

e sostituendo i valori numerici matriciali ottenuti al passo precedente, possiamo ottenere il sistema di equazioni, le quali rappresentano l'insieme delle funzioni di trasferimento caratterizzanti il sistema:

$$\begin{pmatrix} \frac{2.75}{s^2} - \frac{1.749e+13}{2.392e+13 s^2 - 4.398e+12 s^4} \\ \frac{2.75}{s} - \frac{1.749e+13}{2.392e+13 s - 4.398e+12 s^3} \\ -\frac{1.149e+12}{4.398e+12 s^2 - 2.392e+13} \\ -\frac{1.149e+12 s}{4.398e+12 s^2 - 2.392e+13} \end{pmatrix}$$

Si può notare come la prima e la seconda riga della matrice differiscano solo per un fattore derivativo, così come la terza e la quarta riga; questo è ovvio ed atteso in quanto f_2 è la derivata di f_1 e f_4 la derivata di f_3 .

Nello specifico, queste funzionalità di linearizzazione sono state racchiuse all'interno del *live script* Matlab nominato *VAB_dinamica_numerica.mlx*: in questo modo siamo stati in grado di rendere più efficace ed efficiente la prosecuzione della simulazione, non dovendo eseguire ogni volta anche questa parte di calcolo matematico relativo alla modellizzazione e linearizzazione del sistema.

Siamo andati infatti a sfruttare le funzionalità di salvataggio offerte da Matlab per salvare in un file *.mat* l'intero *workspace* (file che abbiamo nominato *WS_VAB_wb_0_5.mat*), in maniera tale da permettere, agli script che ne avessero bisogno, di aprire il workspace e leggere tutte le variabili di interesse.

2.3 Motore

Il passo successivo è stato quello di andare a modellizzare la coppia di motori presenti a bordo dello chassis; questo è stato necessario in quanto si deve tenere conto, in primo luogo, del ritardo che gli attuatori introducono nel sistema, che potrebbe portare il sistema ad essere instabile.

I motori utilizzati per questa applicazione sono motori a corrente continua a magneti permanenti Raw Planet prodotto da Sinobi Serie 75 PX 053: i vari parametri, relativi alle inerzie, coppie nominali, coppie di picco etc. sono stati ricavati dal relativo datasheet.

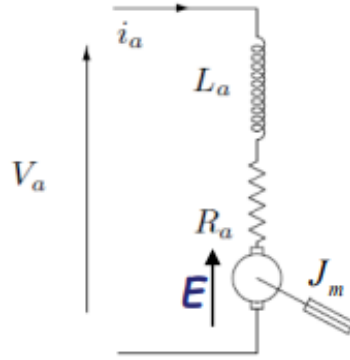


Figure 2.5: Modellizzazione della parte elettrica del motore

In particolare abbiamo portato avanti alcune osservazioni, partendo da quella che è la modellizzazione del motore:

- Da specifiche progettuali, sappiamo che nel motore può scorrere una corrente massima di 20A. Ne segue quindi una limitazione sulla coppia massima esprimibile dal motore stesso, secondo la seguente legge:

$$C_{m,max} = I_{max} \cdot K_{motore} = 20A \cdot 10 \frac{N \cdot m}{A} = 2N \cdot m$$

Questo ci ha portato poi ad inserire all'interno del modello sSimulink, come si vede in figura 2.7, un blocco di saturazione della coppia richiesta dal motore stesso;

- Nella definizione dei parametri caratteristici del motore, è necessario dare spazio anche ad alcune considerazioni su quelli che sono le costanti di tempo che si ricavano dalla modellizzazione della funzione di trasferimento del motore, partendo dal modello elettrico (figura 2.5)

- $\tau_e = \frac{L_a}{R_a} = \text{costante di tempo elettrica}$
- $\tau_m = \frac{J_m + J_r \tau^2}{K_e K_t} = \frac{J_m + J_r}{K^2} = \text{costante di tempo meccanica}$

Infatti, nella funzione di trasferimento che considera in ingresso la tensione di armatura V_a e in uscita la velocità Ω , espressa utilizzando le costanti di tempo sopra riportate, presenta una coppia di poli che possono essere o reali o complessi e coniugati a seconda dei valori assunti da τ_e e τ_m . Nello specifico, come visto nella parte teorica, i poli saranno reali se $\xi = \frac{1}{2} \sqrt{\frac{\tau_m}{\tau_e}} > 1$, ovvero se $\tau_m \gg 4\tau_e$: questa è quindi una condizione che abbiamo verificato, in maniera tale da permettere una semplificazione del modello del sistema nella parte di definizione dei poli. Abbiamo infatti che:

- $\tau_e = 1.22ms$
- $\tau_m = 18.3ms$

da cui si può evincere che la costante di tempo meccanica risulti essere molto maggiore di quella elettrica, il che significa che la dinamica meccanica e quella elettrica sono disaccoppiati in frequenza, potendo così considerarle in maniera separata.

- Nel calcolo della costante di tempo meccanica τ_m abbiamo sviluppato due considerazioni:
 - è necessario considerare anche l'inerzia del carico J_r , ovvero delle ruote ridotte all'albero motore, nel calcolo della costante di tempo meccanica;
 - Le costanti di coppia K_T e elettrica K_E rappresentano gli stessi valori, espressi in unità di misura differenti, che quindi chiamiamo, per semplicità K ;
- Abbiamo anche verificato l'aderenza del modello creato a quanto visto in teoria: sappiamo infatti che, andando a semplificare l'influenza della parte meccanica (quindi come se fosse a rotore bloccato), considerando solamente quella elettrica, avremo un effettivo comportamento come filtro passa basso. Nel modello completo invece, che considera anche la parte meccanica, abbiamo che in bassa frequenza si raggiunge un equilibrio tra la forza contro elettro motrice e la velocità di rotazione del rotore.

Quanto detto è confermato anche dei diagrammi in figura 2.6.

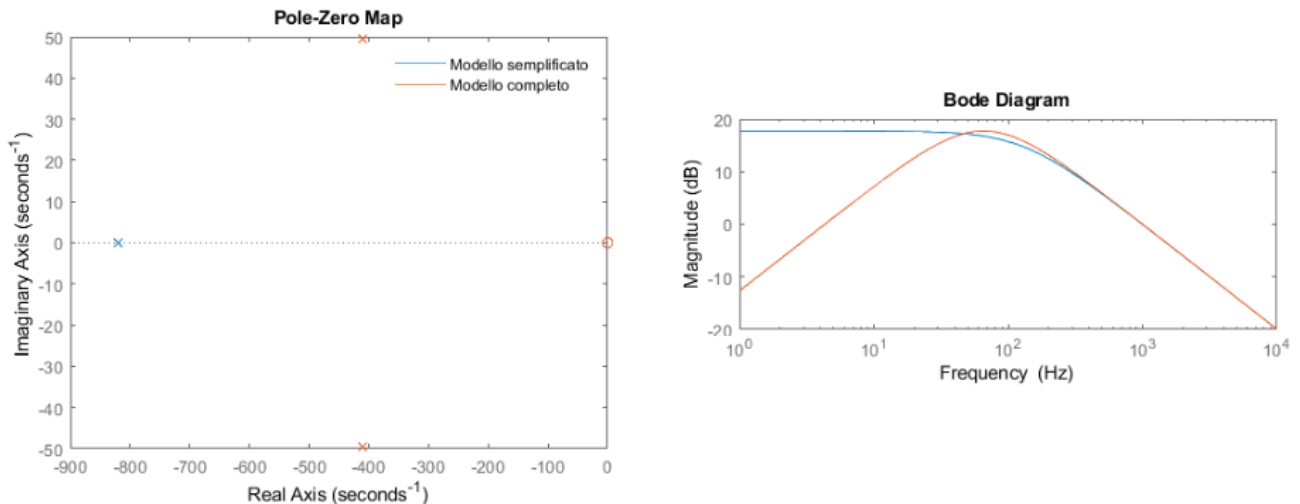


Figure 2.6: Diagrammi di Bode modello del motore in anello aperto

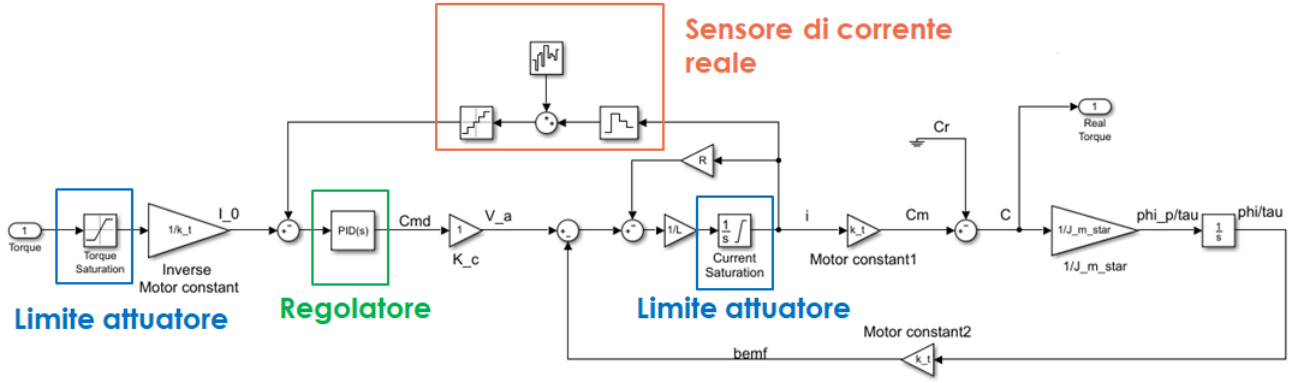


Figure 2.7: Schema a blocchi della modellizzazione del motore

Per comprendere quanto sia influente l'approssimazione $\tau_m \gg 4\tau_e$, possiamo andare ad analizzare quella che è la risposta in frequenza del motore, rappresentandola tramite diagrammi di Bode TODO

In figura 2.7 è rappresentato il modello finale completo del motore, in campo discreto: abbiamo messo in evidenza, tramite alcuni blocchi colorati, alcuni punti principali

- *Limite attuatore*: come evidenziato in precedenza, da specifiche è noto che la corrente massima assorbibile dall'attuatore sia limitata ad un valore max. Ne consegue quindi che, anche la coppia risulti essere limitata: questi due comportamenti sono stati modellizzati con un blocco Simulink di saturazione (nel caso della limitazione di corrente massima, abbiamo unito la saturazione al blocco integrale);
- *Regolatore*: il controllo del motore DC in questione è stato ottenuto tramite una retroazione in corrente che permette quindi di definire un setpoint alla corrente fornita al motore.

Questo si è reso necessario poiché il controllore, attraverso il vettore K e lo stato del sistema, definisce la coppia che il motore dovrebbe erogare: in un motore DC la correlazione tra coppia erogata e corrente esiste ed è ben definita dalla costante K_t . Il controllore è stato realizzato seguendo metodi già noti in letteratura, che cerchiamo di riassumere brevemente di seguito. La funzione di trasferimento che lega V_a ad I_a abbiamo visto essere

$$tf_{VI,complete} = G(s) = \frac{\frac{\tau_m}{R_a} s}{\tau_m \tau_e s^2 + \tau_m s + 1}$$

Volendo andare ad applicare un regolatore di tipo **PI (Proporzionale - Integrale)**, andremo ad inserire il seguente blocco regolatore

$$R_{PI}(s) = K_P + \frac{K_I}{s} = K_P \left(1 + \frac{K_I}{K_P s}\right) \rightarrow \tau_I = \frac{K_P}{K_I}$$

$$\rightarrow K_P \left(1 + \frac{1}{\tau_I s}\right) = K_P \left(\frac{1 + \tau_I s}{\tau_I s}\right) = K_P \frac{K_I}{K_P} \left(\frac{1 + \tau_I s}{s}\right) = k_{gi} \left(\frac{1 + s\tau_i}{s\tau_i}\right)$$

dove $k_{gi} = 2\pi f_c \tau_e R$. Con il comando Matlab *feedback*, siamo quindi stati in grado di ottenere le *f.d.t.* sia semplificate che non, implementando quindi quanto riportato nella figura di dettaglio 2.8.

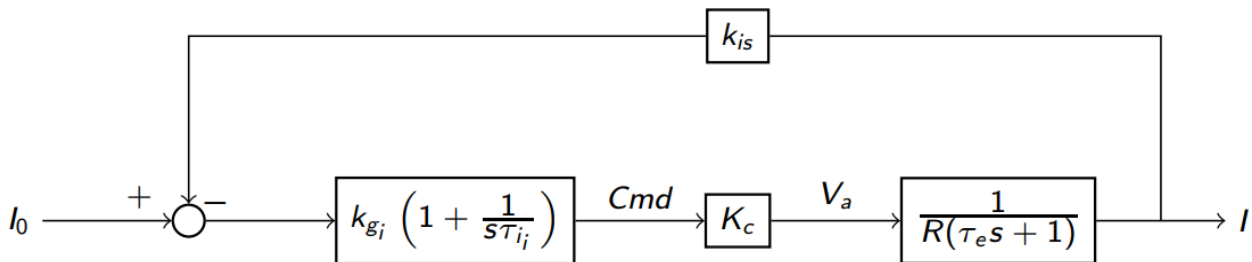


Figure 2.8: Dettaglio del modello del motore

- *Sensore di corrente*: la retroazione in corrente è ovviamente necessaria, essendo il controllo basato sulla corrente assorbita dallo stesso. In particolare, a differenza del modello continuo in cui ci siamo limitati solamente a riportare il segnale di corrente in ingresso, nella modellizzazione discreta siamo andati ad inserire alcuni blocchi *Simulink* per modellizzare l'andamento discreto della grandezza fornita dal sensore, unitamente alla componente di rumore.

3

Controllo

3.1 Introduzione

Per rendere maggiormente leggibile il codice Matlab e, allo stesso tempo, ridurre i tempi necessari per l'esecuzione, abbiamo deciso di dividere le varie funzionalità in diversi *livescript*: in particolar modo la parte relativa allo studio ed alla definizione del controllore siamo andati ad inserirla all'interno del file *GainCalculator.mlx* dove si può vedere che le prime istruzioni vanno a caricare dal workspace, risultante dalla precedente linearizzazione, le matrici A-B-C-D che definiscono il sistema lineare modellizzato.

3.2 Analisi in anello aperto

Prima di procedere con l'individuazione del controllore più adatto per stabilizzare il V.A.B., siamo andati ad effettuare una breve analisi sul sistema ad anello aperto, per ottenere così alcuni spunti sulla correttezza del modello che era stato steso fino a quel punto.

Come noto dalla teoria, il sistema risulta essere asintoticamente stabile se gli autovalori della matrice A risultano avere tutti parte reale negativa; è instabile invece se è presente almeno un autovalore con parte reale strettamente positiva.

Abbiamo quindi calcolato gli autovalori con il comando seguente:

```
eig(A_real)
```

Listing 3.1: Calcolo autovalori matrice A (non simbolica)

La presenza di autovalori con parte reale strettamente positiva è stata confermata anche dalla visualizzazione del luogo delle radici (*rlocus*) del sistema in anello aperto: come si vede in figura 3.1, il polo che si trova nel semi-piano reale positivo non potrà, in alcun modo, essere stabilizzato, ovvero portato nel semi-piano sinistro.

Questa breve analisi ci ha permesso di avere una conferma numerica del fatto ovvio che, senza un controllo, il V.A.B. in esame non riesce a mantenere da solo la posizione verticale.

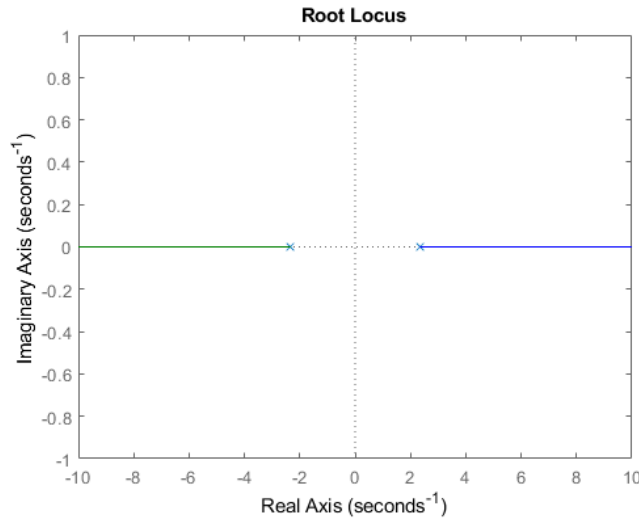


Figure 3.1: Luogo delle radici del V.A.B. modellizzato in anello aperto

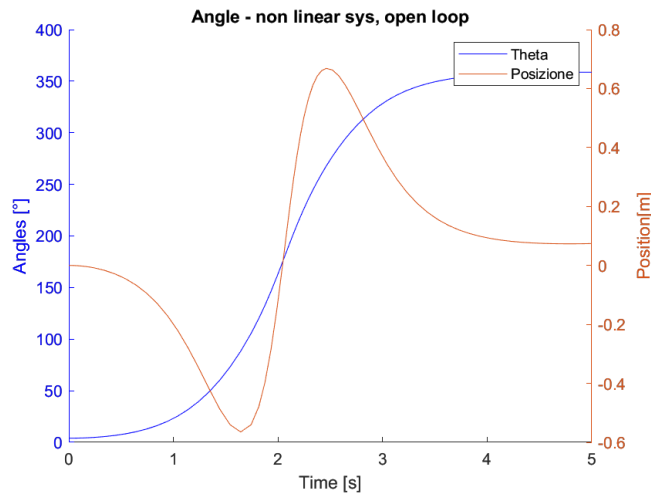


Figure 3.2: Risposta del sistema non lineare in anello aperto

3.3 Controllore

Per quanto riguarda il sistema del veicolo autobilanciato, siamo andati a far riferimento a quella classe di problemi definiti come *problemi di regolazione*, in cui il sistema si presenta inizialmente con una condizione iniziale dello stato non nulla, condizione che si intende di riportare a zero con una velocità di convergenza assegnata.

Nello specifico, questo tipo di problema relativo alla scelta del controllore, è stato risolto mediante l'utilizzo dell'assegnazione degli autovalori ottenuti tramite retroazione dello stato, in cui il moto del sistema è composto completamente dal moto libero che si vuole controllare e annullare in un tempo a piacere.

Il posizionamento dei poli, e quindi la scelta del guadagno del regolatore, va eseguita sul solo sistema lineare: questo a conferma sia di quanto riportato in figura 3.3 (nel blocco di colore blu infatti è riportato il sistema lineare), sia di quanto detto in precedenza in merito al fatto che il controllore venga ad essere definito lavorando sul sistema linearizzato.

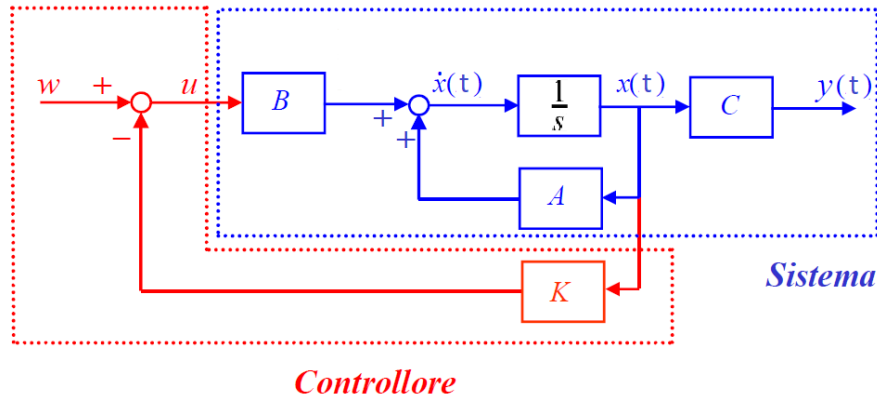


Figure 3.3: Schema concettuale del controllo ([1])

Ciò che è riportato graficamente in figura 3.3, può essere riscritto matematicamente come segue:

$$\begin{cases} \dot{x}(t) = Ax(t) + Bu(t) \\ y(t) = Cx(t) + Du(t) \end{cases}$$

$$u(t) = -Kx(t) + w(t) = \text{legge di controllo}$$

Dunque, sostituendo la legge di controllo nell'equazione di stato del sistema, possiamo ottenere le seguenti equazioni che ci permettono di definire il legame del sistema in anello chiuso con la matrice K:

$$\dot{x}(t) = Ax(t) + Bu(t) = Ax(t) + B(-Kx(t) + w(t))$$

$$= Ax(t) - BKx(t) + Bw(t) = (A - BK)x(t) + Bw(t)$$

$$A_{cl} = A - BK = \text{matrice di stato in anello chiuso (cl} \rightarrow \text{closed loop)}$$

3.3.1 Posizionamento dei poli continui

Per scegliere il valore da assegnare a K, e quindi definire il guadagno del controllore, è necessario scegliere la posizione desiderata dei poli, in base a quelle che sono le specifiche di velocità desiderate.

$$G(S) = \frac{\omega_n^2}{s^2 + 2\xi\omega_n s + \omega_n^2}$$

$$\xi = 0.7$$

$$\omega_n = 2\pi f_{\text{propria}}$$

Dovendo posizionare due coppie di poli complessi coniugati si sono scelte due frequenze ad una decade di distanza, in maniera tale da poter considerare le variabili che sono controllate disaccoppiate in frequenza: essendo il limite di banda (ovvero entro quali limiti il segnale passa senza essere attenuato e/o modificato) del sistema interno molto più alto del limite di banda del sistema esterno, possiamo considerarli disaccoppiati in frequenza.

$$f_{\text{propria}\theta} = 0.2 \text{ Hz}$$

$$f_{\text{propria}\phi} = 0.02 \text{ Hz}$$

Nello specifico, una prima osservazione, è il fatto che il polo più veloce (ovvero quello con frequenza pari a 0.2 Hz) è stato assegnato alla parte relativa al controllo dell'angolo θ , essendo che è auspicabile un controllo maggiormente reattivo per quanto riguarda la stabilizzazione della base, piuttosto che il rapido raggiungimento del set point di posizione spaziale (e quindi angolare ϕ delle ruote).

Una seconda osservazione è che, per motivi esterni, il motore risulta avere una coppia massima molto limitata (per via delle limitazioni di corrente): è stato dunque necessario posizionare i poli ad una frequenza tale che permettessero di non saturare per molto tempo la coppia fornita dal motore. Questa scelta ha inevitabilmente rallentato la risposta del sistema.

Definita quindi la frequenza a cui posizionare i poli, siamo andati a risolvere l'equazione al denominatore della forma generica della f.d.t ($G(s)$) con due poli complessi: la risoluzione di questa equazione, come è noto dalla teoria, permette di ottenere i valori della variabile di Laplace s che azzerano il denominatore stesso, che corrisponde a:

$$s^2 + 2\xi s w_n + w_n^2 = 0$$

Nella nostra simulazione i valori numerici ottenuti per i poli sono stati i seguenti:

$$polo_\theta = \begin{pmatrix} -\frac{7\pi}{250} + \frac{\pi\sqrt{51}i}{250} \\ -\frac{7\pi}{250} - \frac{\pi\sqrt{51}i}{250} \end{pmatrix}$$

$$polo_\phi = \begin{pmatrix} -\frac{7\pi}{25} + \frac{\pi\sqrt{51}i}{25} \\ -\frac{7\pi}{25} - \frac{\pi\sqrt{51}i}{25} \end{pmatrix}$$

Con il comando *place* di Matlab si ottiene dunque:

$$K = [-0.0023, -0.0278, -28.1397, -7.7022]$$

Ricordando a questo punto quanto detto nella prima parte di questo capitolo in merito alla definizione della matrice A in anello chiuso, siamo andati a definirla appunto come

$$A_{cl} = A - BK$$

utilizzando il valore di K appena trovato: una volta trovata la nuova matrice A , siamo andati a ri-effettuare le stesse analisi portate avanti anche per il sistema in anello aperto.

Come si vede in figura 3.4, grazie al posizionamento degli autovalori, siamo stati in grado di spostare i nuovi poli nel semi-piano sinistro: questo significa quindi che siamo riusciti a stabilizzare il nostro sistema dinamico.

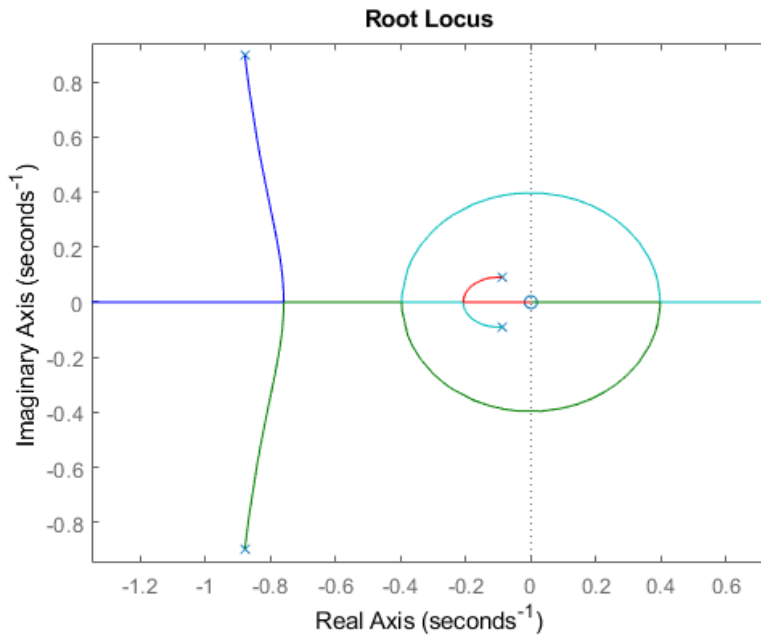


Figure 3.4: Root locus del sistema in anello chiuso

3.3.2 Setpoint di velocità

Oltre al controllo del moto libero del sistema è stata implementata anche la possibilità di inserire un setpoint sulla velocità a fine transitorio del Segway. In questo modo si apre la possibilità per l'utente finale di impostare la velocità desiderata e di mantenerla nel tempo nonostante i vari disturbi che in un sistema reale agiranno sulla macchina (vento, salita, discesa..)

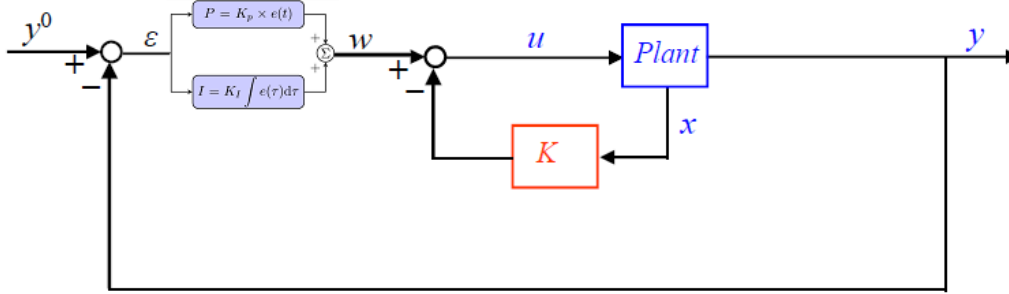


Figure 3.5: chiusura dell'anello di controllo di $\dot{\phi}$ con un controllore I [1]

In figura 3.5 si nota come l'anello di $\dot{\phi}$ sia esterno rispetto all'anello di retroazione dello stato; solitamente si procede dunque nel progettare il controllore avendo cura di lasciare una decade di spazio tra la frequenza del polo dell'integratore e la coppia di poli più lenta del controllore della retroazione dello stato.

In questo caso, per ragioni puramente pratiche, non è stato possibile: il sistema presenta una risposta particolarmente lenta agli input viste le limitazioni di coppia (imposte dai limiti di corrente definiti in fase di specifica del progetto) di cui si approfondirà successivamente.

Sarebbe quindi stato necessario quindi troppo tempo per raggiungere il setpoint di velocità se si fosse proceduto a lasciare una decade di distanza. Si è invece proceduto a posizionare il polo dell'integratore tramite la funzione di Tuning offerta da Matlab Simulink stesso:

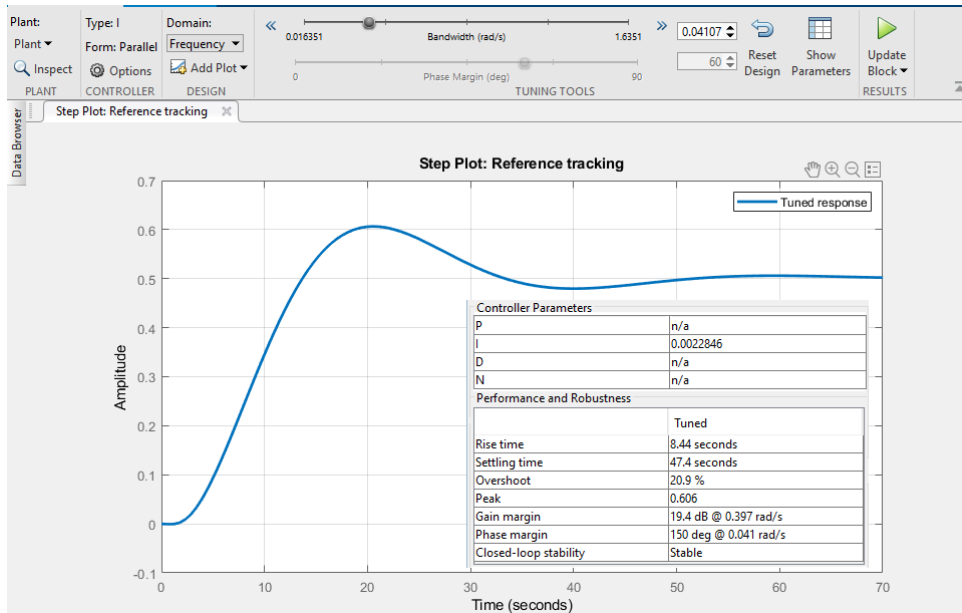


Figure 3.6: Taratura del controllore

Il tuning è stato effettuato sul sistema lineare in quanto Matlab richiede un sistema lineare per questo tipo di tecniche.

Da notare inoltre, in figura 3.6 che la frequenza $f_3 = \frac{0.041 \text{ rad/s}}{2\pi} = 0.0065 \text{ Hz}$ che è circa $\frac{1}{3} f_{\text{propria } \phi}$.

Sempre in figura 3.6 si osserva che, nonostante non sia stata rispettato la decade di distanza, il sistema sia comunque molto lento e arrivi a regime in circa 70 s.

TODO: perchè arriva a 0.5? per la retroazione interna?

3.4 OPC - UA

Per quanto riguarda la parte di controllo, il sistema presenta un controllore centralizzato, rappresentato da un Raspberry: nello specifico ad esso sono affidate delle mansioni ben specifiche, tutte ovviamente volte al controllo e alla stabilizzazione del *veicolo auto bilanciato*.

In questa fase dello sviluppo del progetto, siamo andati ad implementare parte del codice che verrà installato, in un secondo momento, a bordo del Raspberry: esso infatti svolge, all'interno del sistema (come si vede in figura 3.7) una comunicazione a due direzioni, che ne determinano due comportamenti differenti:

- Come **server** per la parte di comunicazione *OPC-UA* (per il settaggio dei guadagni) nei confronti di un client (ovvero dell'utente che intende settare i parametri del controllore);
- Come **master** nei confronti della sezione di sensoristica a bordo del V.A.B. (per quanto riguarda invece la gestione dell'algoritmo di controllo);

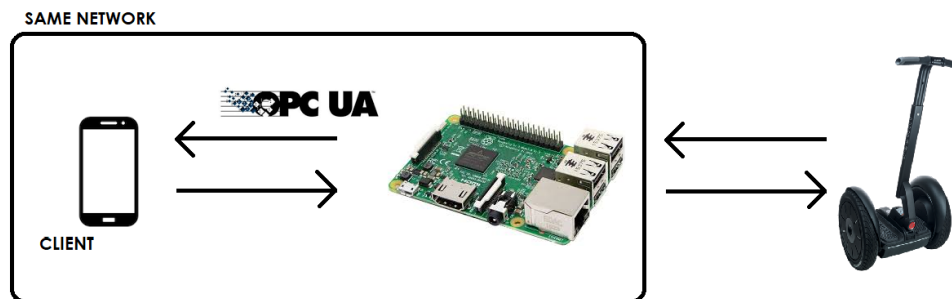


Figure 3.7: Schema di massima dell'utilizzo di Raspberry Pi 3

In questa fase abbiamo quindi sviluppato la parte relativa all'utilizzo di *Raspberry Pi 3* come *server OPCUA*: in seguito, nella sezione 3.6, ci occuperemo di andare a rendere Raspberry un controllore Real-Time, tramite il framework *Xenomai*.

3.4.1 Idea base OPC-UA

L'*Open Platform Communications Unified Architecture* (OPC UA) è un protocollo di comunicazione automatico per l'automazione industriale: OPC UA sostituisce il protocollo *OPC Classic*, conservando tutte le funzionalità del predecessore.

Poiché *OPC Classic* è stato costruito su una tecnologia Microsoft detta modello a oggetti per componenti distribuiti, risulta essere vincolato a Microsoft (caratteristica è diventata sempre più limitante).

OPC UA invece risulta completamente interoperabile tra i diversi sistemi operativi usati, diventando compatibile, oltre che con Windows, anche con tecnologie industriali come i PLC, Linux, iOS e anche sistemi operativi per dispositivi mobili come Android.

Queste caratteristiche di **interoperabilità** sono state sfruttate al massimo in questo contesto, potendo così creare, in maniera semplice e veloce, una comunicazione tra differenti tipologie e famiglie di dispositivi.

3.5 OPC-UA e V.A.B.

Nel contesto del progetto del *veicolo auto bilanciato*, siamo andati ad utilizzare il protocollo di comunicazione *OPC-UA* come supporto per il tuning dei parametri relativi al **gain** del controllore, ovvero ai parametri del vettore K , che abbiamo chiamato (all'interno dello script di Python):

- **K_phi**
- **K_phi_p**
- **K_theta**
- **K_theta_p**

Nello specifico, lo scambio di parametri tra server e controllore avviene tramite un file di testo *.txt* che permette, in maniera semplice e immediata, di implementare uno scambio di informazioni tra il server *OPC-UA* strutturato in Python e l'ambiente *Real-time* introdotto a bordo del controllore *Raspberry Pi 3*.

In particolare abbiamo due files:

- **Un file temporaneo** ("*GainParametersToController.txt*") utilizzato come pipeline per il passaggio dei parametri tra server e controllore. Questo risulta essere un file temporaneo che viene ad essere cancellato e ricreato ogni qualvolta che il server viene spento e successivamente riaccessso. Nello specifico, ad ogni riaccensione, i valori iniziali di questo file, vengono settati con gli stessi valori presenti nel file *definitivo* (qualora quest'ultimo non esistesse, si procede con un'inizializzazione dei parametri a 0);
- **Un file definitivo** ("*GainParametersConfirmed.txt*") il quale invece viene creato una e una sola volta e sul quale poi vengono salvati i parametri che saranno poi letti all'accensione successiva del server ed utilizzati come parametri iniziali per il controllore.

Questi file possono essere settati con i parametri accettati nel server che sono visibili in figura 3.8; nello specifico:

- **Submit change to controller** permette di scrivere i valori dei gains sul file "*GainParametersToController.txt*";
- **Store definitively in file** permette di scrivere i valori dei gains sul file "*GainParametersConfirmed.txt*";
- **SHUT DOWN SERVER** permette invece di spegnere il server e di cancellare il file temporaneo.

All'interno dell'Appendice A, abbiamo racchiuso in figura A.1, tramite diagramma di flusso, il funzionamento di massima del codice lato server: codice che siamo andati a testare utilizzando un'apposita app per smartphone Android (OPC-UA Android client).

Per quanto riguarda invece l'effettiva implementazione, sempre in Appendice A, abbiamo riportato l'intero codice prodotto per la gestione della comunicazione OPC-UA.

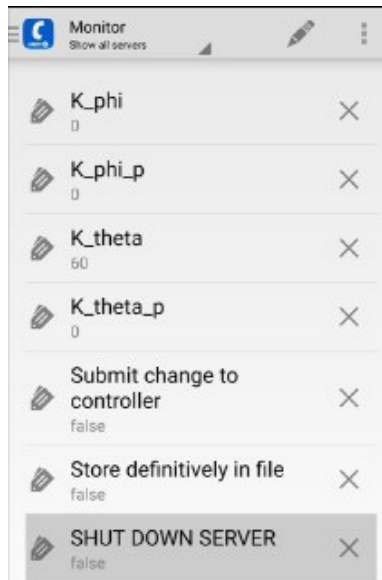


Figure 3.8: Parametri impostabili lato client

3.6 RTOS - XENOMAI

Nella parte di controllo di sistemi in generale, risulta essere sempre necessario andare a lavorare in *Real-Time*, ovvero il sistema richiede che il calcolatore sia in grado di produrre computazioni corrette andando allo stesso tempo a rispettare delle *deadline* temporali, ovvero dei vincoli che validano o meno il risultato prodotto.

Come ben noto nell'ambiente informatico, *Linux*, che rappresenta l'*OS* installato a bordo di Raspberry, risulta essere un sistema operativo **non RTOS** (Real Time Operative System): è stato quindi necessario andare ad improntare un approccio che permesso di renderlo tale.

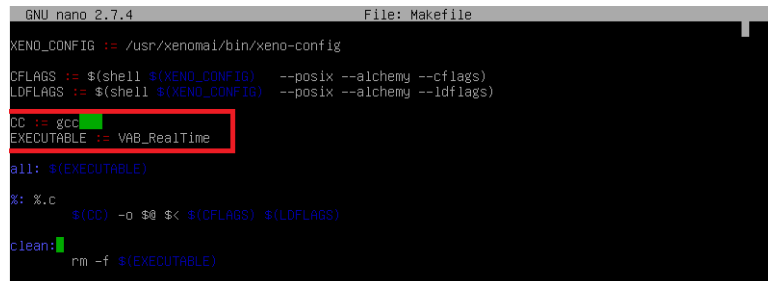
In questo contesto è risultato essere di fondamentale importanza il software *Xenomai*, un progetto open source creato per fornire un framework real-time per le piattaforme Linux, con l'obiettivo di aiutare la migrazione di progetti in ambito industriale da sistemi real-time proprietari a sistemi diffusi come Linux.

Il cuore delle funzionalità offerte da Xenomai, sta nella praticità e facilità con cui mette a disposizione delle API di tipo real-time, occupandosi anche di garantire quelli che sono i limiti e vincoli temporali.

Di seguito una breve descrizione degli step principali seguiti per quanto concerne la parte di adattamento del controllore Raspberry a RTOS.

- Come prima cosa siamo andati a rivedere i files *.cpp* contenente la logica di controllo, ritoccando la parte relativa alla lettura dei dati dal file, andando ad utilizzare le API presenti in ambiente *c* invece di quelle relative a *cpp*. Abbiamo quindi provato e testato la lettura da file tramite il tool DevC++, facilitando così il testing e il debug;
- Siamo andati ad installare una macchina virtuale con a bordo **debian 9.8.0** e la patch **Xenomai 3.0.8** ([2]);
- Essendo debian un OS interamente fruibile da linea di comando, siamo andati a prendere confidenza con l'ambiente, cercando di capire l'organizzazione e chiamando gli update/upgrade necessari;
- Prima di andare ad importare il file completo (di cui abbiamo parlato al punto iniziale), siamo andati ad eseguire alcuni esempi iniziali (una sorta di HelloWorld):
 - Abbiamo per primo cosa studiato questo approccio iniziale (link)
 - Siamo andati ad impostare il funzionamento di un task (dummy) periodico, come se si trattasse appunto del main loop di un sistema Real-Time;

- Lo step successivo è stato quello di importare il codice scritto e testato in ambiente DevC++ in Xenomai, provandolo e introducendo alcuni dettagli aggiuntivi, come presentato nei punti seguenti:
 - Come primo step risulta essere necessario fornire al framework Xenomai il nome di quale file con estensione .c andare a compilare e successivamente, incapsulare in un framework real-time



```
GNU nano 2.7.4 File: Makefile
XENO_CONFIG := /usr/xenomai/bin/xeno-config
CFLAGS := $(shell $(XENO_CONFIG) --posix --alchemy --cflags)
LDLAGS := $(shell $(XENO_CONFIG) --posix --alchemy --ldflags)
CC := gcc
EXECUTABLE := VAB_RealTime
all: $(EXECUTABLE)
%.c:
$(CC) -o $@ $< $(CFLAGS) $(LDLAGS)
clean:
rm -f $(EXECUTABLE)
```

Figure 3.9: Per specificare quale file .c compilare

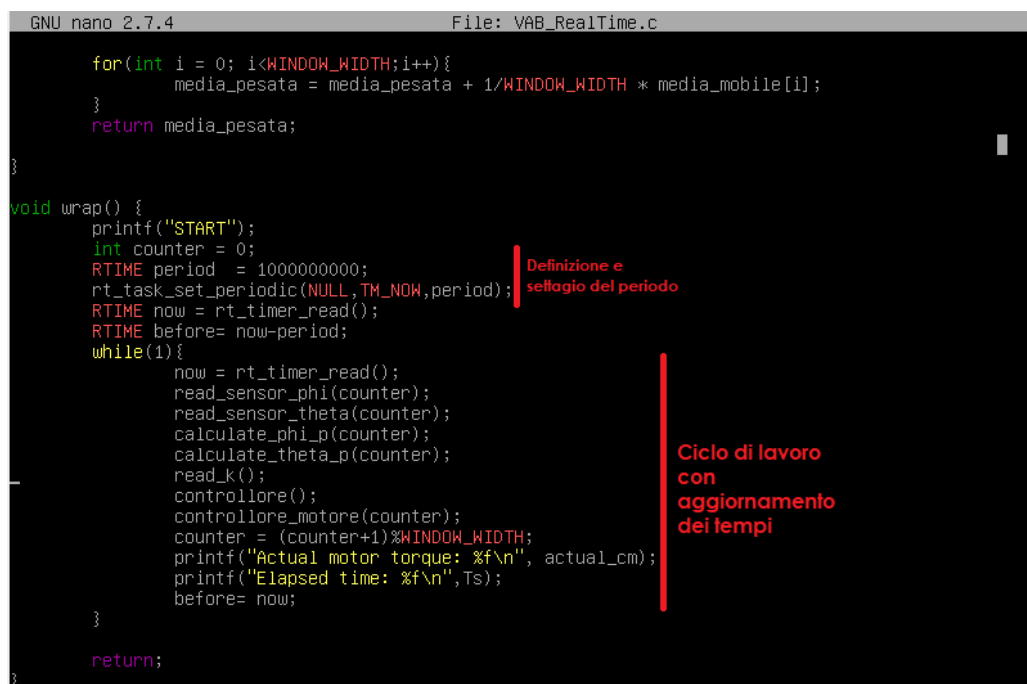
- Abbiamo specificato quello che è il periodo di funzionamento tramite i seguenti comandi:

```
RTIME period = 1000000000; % Espresso in [ns]
rt_task_set_periodic(NULL, TM_NOW, period);
```

- All'interno del ciclo while, abbiamo definito un continuo update dei parametri temporali di nostro interesse.

Nello specifico ad inizio ciclo andiamo a sovrascrivere costantemente il parametro *now* mentre, nella parte finale del ciclo stesso, abbiamo impostato un update del parametro *before*.

Il salvataggio di queste variabili permette di facilitare, in qualsiasi punto del codice, l'accesso ai parametri temporali per eventuali necessità di calcolo, come nella definizione dell'intervallo di tempo su cui andare a realizzare i vari calcoli derivativi necessari per il corretto controllo;



```
GNU nano 2.7.4 File: VAB_RealTime.c

for(int i = 0; i < WINDOW_WIDTH; i++){
    media_pesata = media_pesata + 1/WINDOW_WIDTH * media_mobile[i];
}
return media_pesata;
}

void wrap() {
    printf("START");
    int counter = 0;
    RTIME period = 1000000000;
    rt_task_set_periodic(NULL, TM_NOW, period);
    RTIME now = rt_timer_read();
    RTIME before = now - period;
    while(1){
        now = rt_timer_read();
        read_sensor_phi(counter);
        read_sensor_theta(counter);
        calculate_phi_p(counter);
        calculate_theta_p(counter);
        read_k();
        controllatore();
        controllatore_motore(counter);
        counter = (counter+1)%WINDOW_WIDTH;
        printf("Actual motor torque: %f\n", actual_cm);
        printf("Elapsed time: %f\n", Ts);
        before = now;
    }
    return;
}
```

Definizione e
setttaggio del periodo

Ciclo di lavoro
con
aggiornamento
dei tempi

Figure 3.10: Definizione e temporizzazione del ciclo WHILE

- Come si vede anche in figura 3.10, nel ciclo principale abbiamo messo in successione questi blocchi (eseguiti a frequenza fissa):
 - * Lettura dei parametri da files;
 - * Lettura dei valori misurati della parte sensoristica;
 - * Elaborazione dei dati rilevati e algoritmo di controllo;
 - * Produzione dei risultati di interesse
 - In figura 3.11 si può vedere come, ogni ciclo di lettura del file con i parametri, che corrisponde ad un ciclo macchina, presenti un tempo fisso.
- Infatti, con il termine "*Elapsed time*" siamo andati a stampare il *delta* temporale, il quale rappresenta il tempo occupato da un singolo ciclo macchina;

```

Gain value parsed: -> K0 = 0.000000
Gain value read: -> K1 = 0
Gain value parsed: -> K1 = 0.000000
Gain value read: -> K2 = 60
Gain value parsed: -> K2 = 60.000000
Gain value read: -> K3 = 0
Gain value parsed: -> K3 = 0.000000
Actual motor torque: -130.967766
Elapsed time: 0.001000
tempGainFile is open
Gain value read: -> K0 = 0
Gain value parsed: -> K0 = 0.000000
Gain value read: -> K1 = 0
Gain value parsed: -> K1 = 0.000000
Gain value read: -> K2 = 60
Gain value parsed: -> K2 = 60.000000
Gain value read: -> K3 = 0
Gain value parsed: -> K3 = 0.000000
Actual motor torque: -131.039428
Elapsed time: 0.001000
tempGainFile is open
Gain value read: -> K0 = 0
Gain value parsed: -> K0 = 0.000000
Gain value read: -> K1 = 0
Gain value parsed: -> K1 = 0.000000
Gain value read: -> K2 = 60
Gain value parsed: -> K2 = 60.000000
Gain value read: -> K3 = 0
Gain value parsed: -> K3 = 0.000000
Actual motor torque: -131.111109
Elapsed time: 0.001000
root@xenomai308:/home/Laboratorio_SistemiMeccatroniciIII/functions/C-XENOMAI#

```

Figure 3.11: Esecuzione a tempo fissato (1 ms) delle funzioni di lettura e calcolo

- La gestione del thread relativo al server *OPC-UA* non è ancora stato gestito: esso sarà ovviamente lasciato in secondo piano rispetto al thread di lavoro principale temporizzato alla frequenza specifica;

Simulink

4.1 Introduzione

Data la straordinarietà degli eventi che erano in corso dal punto di vista sanitario nel nostro paese si è reso necessario svolgere gran parte del lavoro in modalità a distanza e quindi senza la possibilità di testare il modello matematico appena ottenuto e i successivi risultati dovuti all'azione di controllo sul V.A.B.

Il lavoro quindi è stato svolto per la maggior parte sfruttando il tool *Simulink* di Matlab il quale rappresenta, in poche parole, un risolutore di equazioni differenziali, le quali rappresentano la descrizione risultante del sistema che siamo andati a modellizzare.

Abbiamo così adottato una metodologia di lavoro basata su prototipi sempre più simili a quello che dovrebbe essere il sistema reale, lavorando step by step, partendo da modelli più semplici e aggiungendo, via via, features più complesse e ricercate, in maniera da rendere il più aderente possibile tale modello al funzionamento reale.

Si va ora ad analizzare la risposta del sistema al controllo ottenuto nei punti precedenti, per assicurarci, che quanto scritto sopra valga oltre che nella teoria anche nella pratica:

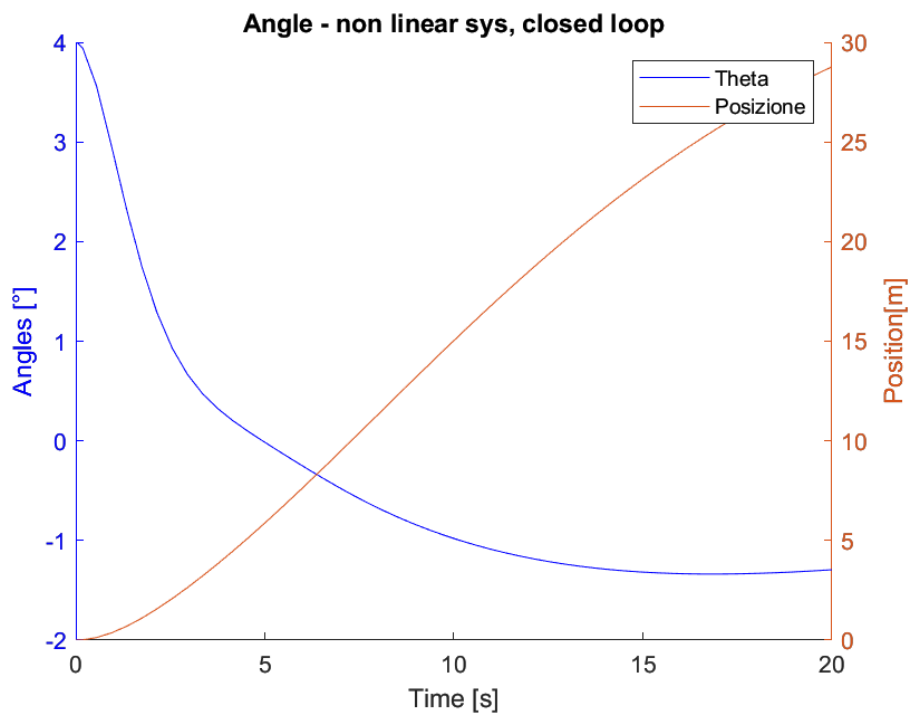


Figure 4.1: Risposta del sistema non lineare in anello chiuso

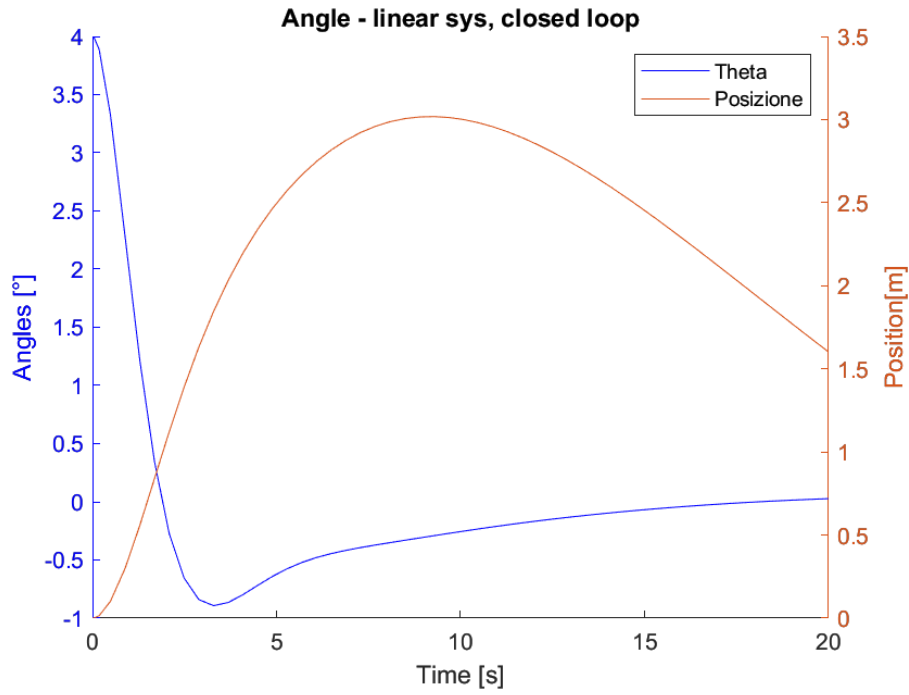


Figure 4.2: Risposta del sistema lineare in anello chiuso

TODO: come spieghiamo la differenza? rifare la simulazione?

4.2 Simulazione del sistema non lineare

Il primo compito che abbiamo risolto è stato quello di implementare le equazioni differenziali ottenute nel capitolo precedente:

- $\ddot{\phi} = f_{\ddot{\phi}}(M_c, \theta, \dot{\theta}, C_m)$
- $\ddot{\theta} = f_{\ddot{\theta}}(M_c, \theta, \dot{\theta}, C_m)$

Dove M_c sarebbe la massa del passeggero, θ e $\dot{\theta}$ lo stato del sistema e C_m la coppia erogata dal motore. Si può notare come entrambe le equazioni differenziali siano indipendenti dalla coordinata libera ϕ .

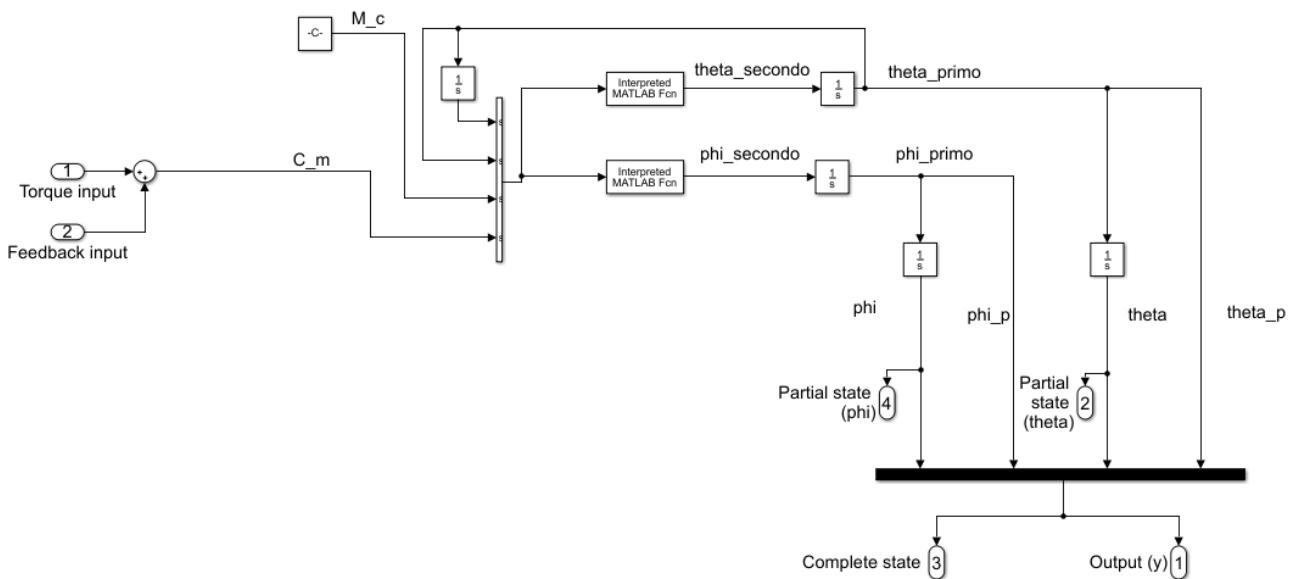


Figure 4.3: Implementazione simulink delle equazioni differenziali

In Fig.4.3 le *interpreted function* altro non sono che $f_{\ddot{\phi}}$ e $f_{\ddot{\theta}}$. A valle di esse sono presenti degli integratori che permettono di ottenere lo stato x completo del sistema. Si può facilmente notare come $\dot{\theta}$ e θ siano collegate direttamente all'input delle *interpreted function*. Si è dunque proceduto a simulare il sistema per verificare la bontà di quanto ottenuto; in particolare, il sistema in anello aperto, dovrebbe oscillare all'infinito vista la mancanza di attriti nel modello.

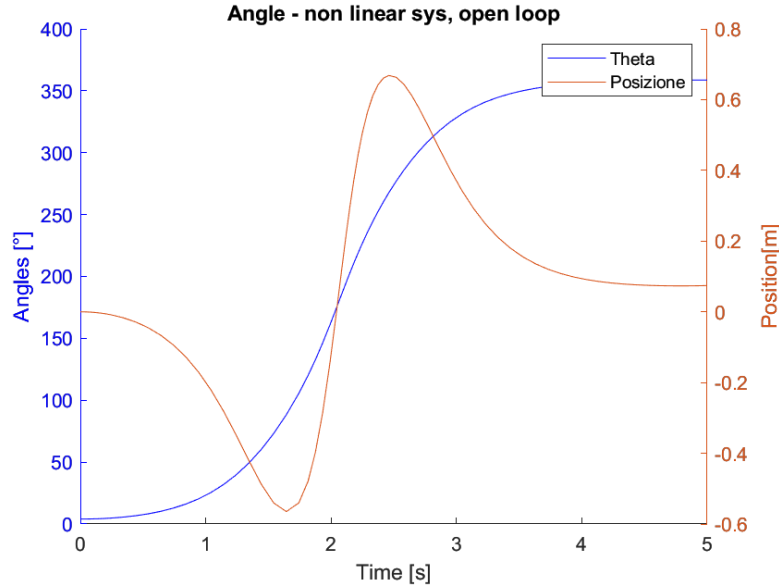


Figure 4.4: Risposta in anello aperto del sistema reale

La simulazione, il cui risultato è riportato in Fig.4.4 è stata svolta per 5 secondi e con un angolo iniziale di 4° ; il grafico mostra dunque l'andamento di θ nel tempo e della posizione che in termini matematici si esprime come $posizione = \phi \cdot r_{ruota}$

4.3 Simulazione del sistema lineare

Si è inoltre creato un altro modello sfruttando il sistema lineare ottenuto prima con l'obbiettivo di semplificare il problema e di velocizzare le simulazioni con lo scopo di testare rapidamente nuove tecniche di controllo che se avessero dato esito positivo sul modello lineare sarebbero poi state testate sul simulink che imita il comportamento reale del sistema.

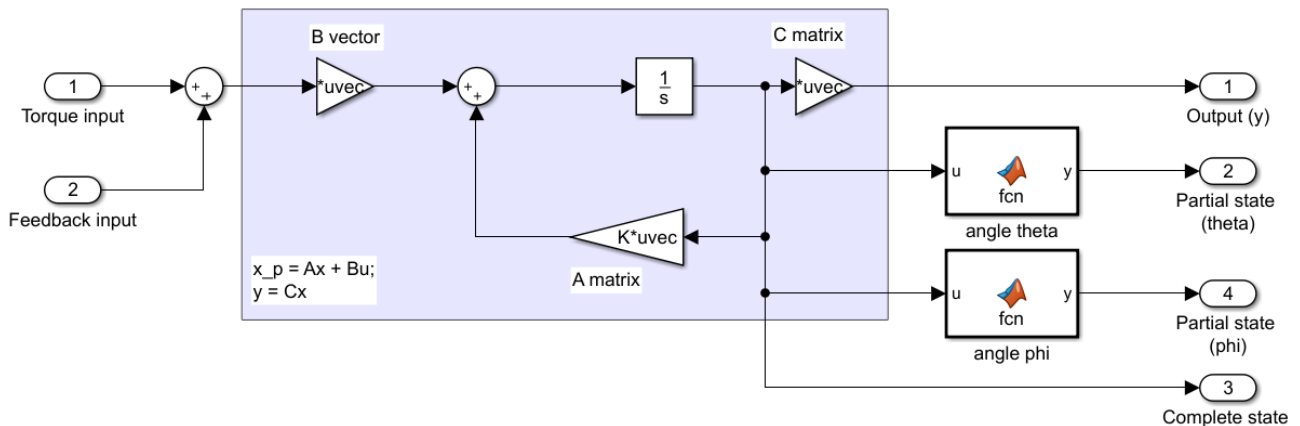


Figure 4.5: Implementazione simulink del sistema linearizzato

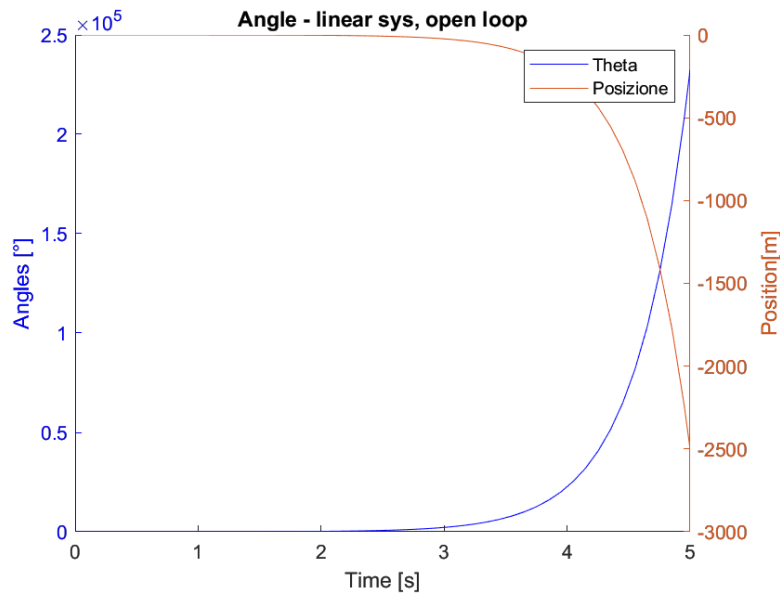


Figure 4.6: Risposta in anello aperto del sistema lineare

In questo caso, la simulazione in anello aperto mostra che il sistema diverge; questo perché la linearizzazione ha senso attorno al punto di equilibrio da cui è stata ottenuta, distante da quel punto il sistema lineare non approssima più il sistema reale ed anche un eventuale controllo ottenuto da esso non garantisce buone performance distante da quel punto. Si nota, in Fig.4.6 che il punto di partenza è 4° e il sistema, lineare, diverga quasi immediatamente; la differenza con la risposta del sistema non lineare in Fig.4.4

Come già verificato in precedenza (sezione 3.2) ci sono due poli reali, uno negativo e uno positivo; questo dimostra come il sistema sia instabile in anello aperto e necessiti di controllo.

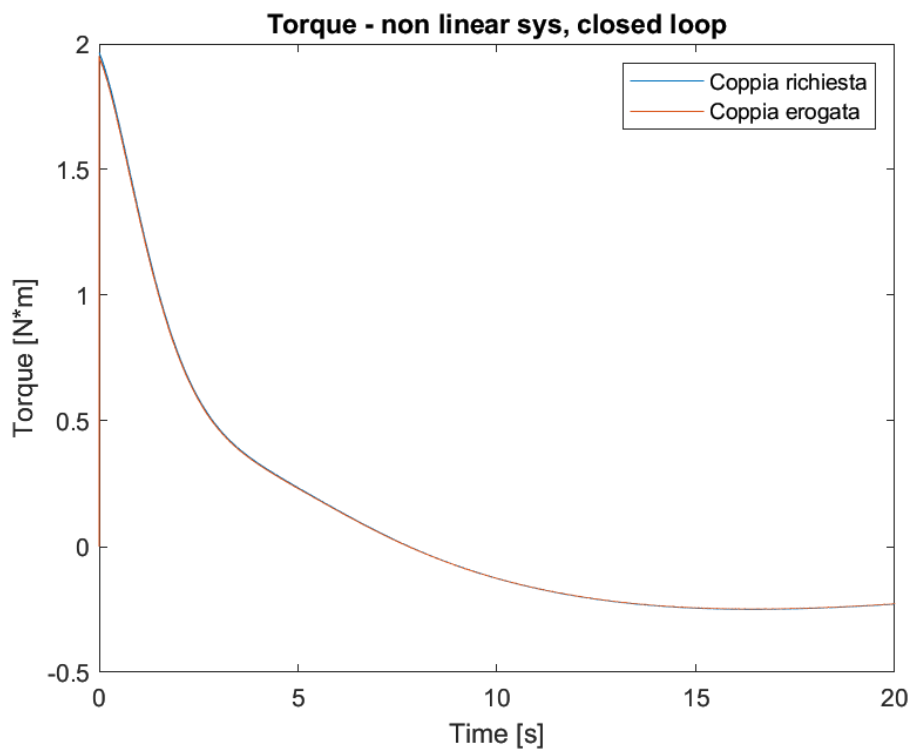


Figure 4.7: Transitorio del motore

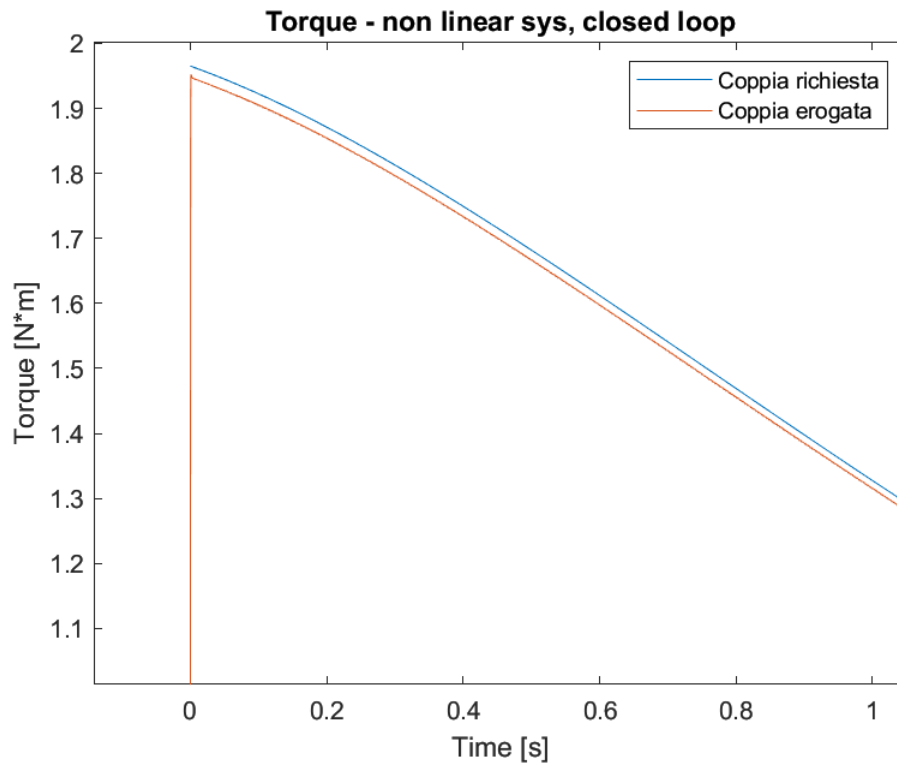


Figure 4.8: Zoom del grafico in figura Fig.4.7

Come si può notare il picco di coppia massimo è minore di 2 Nm, come da limitazioni imposte dal modello del motore.

Un esempio in cui la coppia richiesta supera i $2N \cdot m$ è il seguente:

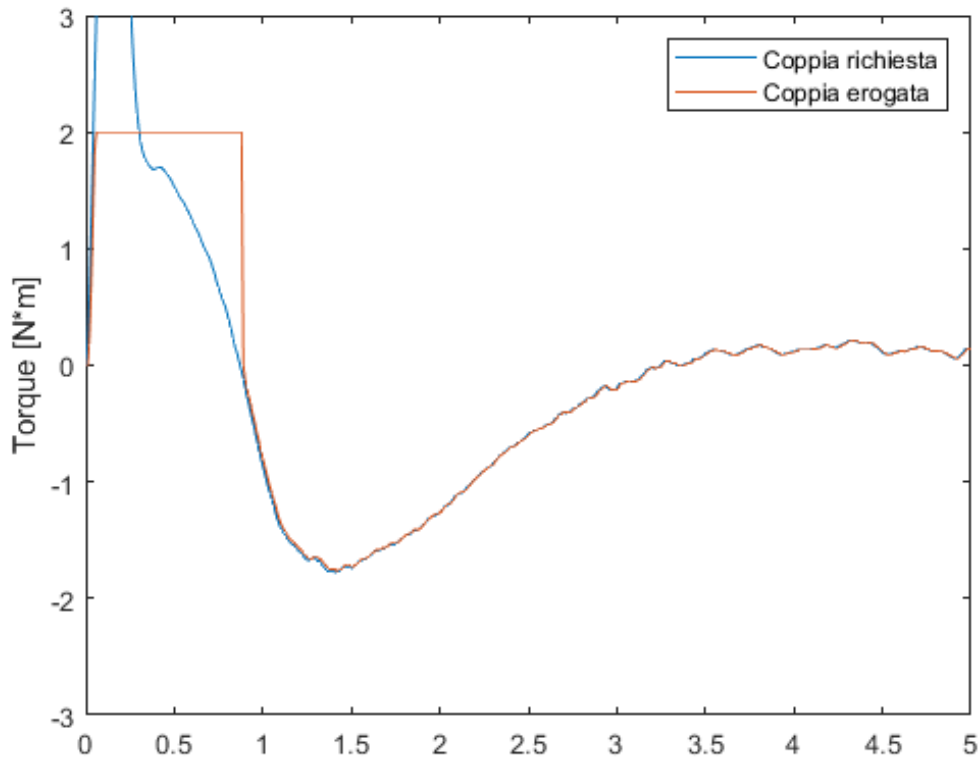


Figure 4.9: Clamp della coppia erogata da parte del motore

In Fig.4.9 è anche possibile notare come l'assenza del blocchetto denominato *Torque saturation*, presente in Fig.2.7, saturo l'azione integrale dell'attuatore e inserisce un ritardo non secondario nell'azione

di controllo.

4.4 Modello complessivo (al momento)

L'obiettivo di questo paragrafo è quello di fare il punto della situazione del sistema sviluppato fino a questo punto e di sviluppare alcune considerazioni sul lavoro fatto.

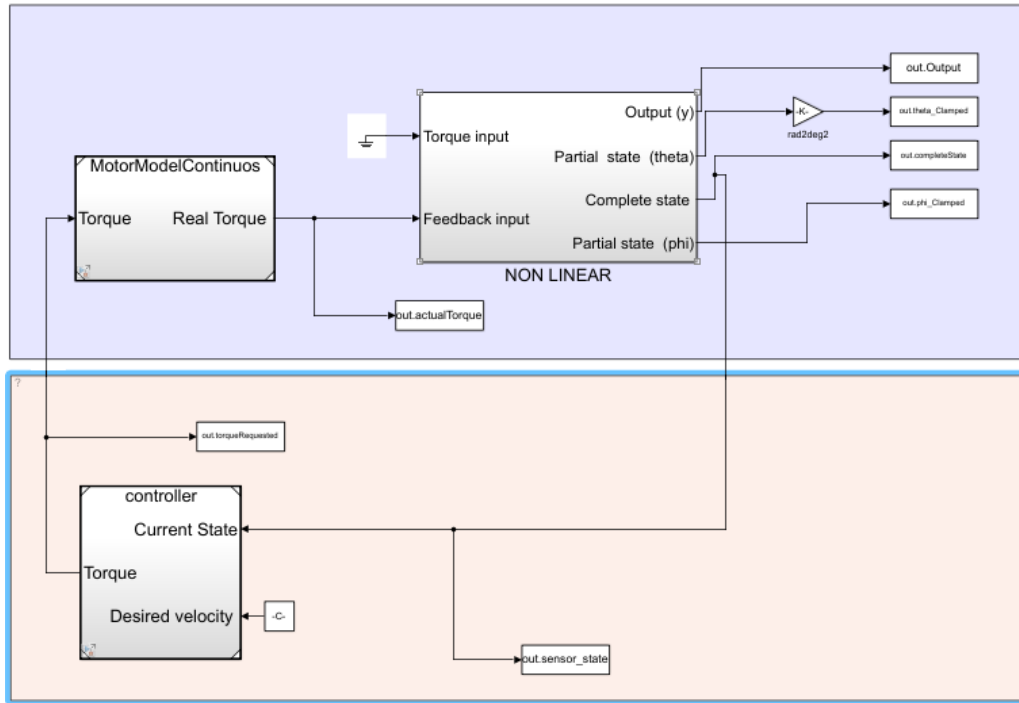


Figure 4.10: Simulink

Come si osserva in Fig.4.10 il sistema al momento comprende tre blocchi:

- Il blocco *non linear*: rappresenta quello che nella realtà sarebbe il sistema reale; si occupa durante la simulazione, dati gli input, di restituire output simili a quelli che si avrebbero in laboratorio utilizzando la macchina vera e propria.
- Il blocco *MotorModelContinuos*: si occupa di simulare la presenza e i transitori dovuti ai motori che nella realtà sono posti a bordo dello chassis.
- Il blocco *controller*: è, dei tre, l'unico blocco che effettivamente dovrebbe essere implementato su un calcolatore. Con i valori simulati dai due blocchi di cui sopra calcola il valore di coppia per il controllo e lo fornisce indietro ai suddetti blocchi per completare la retroazione.

Il sistema al momento è, in linea teorica, a tempo continuo. Nella realtà in un computer e in particolar modo su Simulink la possibilità di far operare i blocchi che simulano il sistema reale a tempo continuo è preclusa. Si è scelto dunque, per quanto fatto finora, di lasciare scegliere al software di Simulink il passo della simulazione ed in particolar modo utilizzare un passo variabile. Questo permette, nei punti in cui le variazioni sono spiccate (ad esempio quando il sistema si avvia) di utilizzare un passo di simulazione anche dell'ordine dei nanosecondi che approssima quasi perfettamente l'esecuzione a tempo continuo.

4.5 Discretizzazione

Perché discretizzare? Come già detto sopra, il controllore è necessario che sia implementato su di un calcolatore, venendo pertanto eseguito a tempo discreto.

Per tenere conto di questa caratteristica è necessario che essa venga modellizzata in quale modo, rendendo il controllore in grado di acquisire gli input provenienti dal sistema a tempo discreto:

- all'ingresso del blocco *controller* è stato posto un *Sample and Holder*; questo componente si occupa di acquisire ad ogni tempo di sampling T_s , il valore in input, campionandolo e mantenendolo inalterato fino alla lettura successiva;
- all'uscita del *controller* si è posizionato un altro *Sample and Holder* per le stesse ragioni;
- il *controller* è stato trasformato a tempo discreto; TODO inseriamo la cosa dei poli discreti o diciamo che sono uguali essendo un proporzionale? A differenza del controllore della retroazione dello stato, il controllore che chiude la retroazione in velocità va ricalcolato tenendo conto del T_s ; fortunatamente Simulink fa da solo questa conversione se si setta il blocco simulink *Controllore PID* come integratore a tempo discreto fornendo il T_s e la costante moltiplicativa;
- Tutte le *grandezze fisiche* misurate sono state campionate e rese rumorose con appositi blocchi Simulink;

Il controllore a tempo discreto ha dunque questo aspetto nel modello simulink implementato:

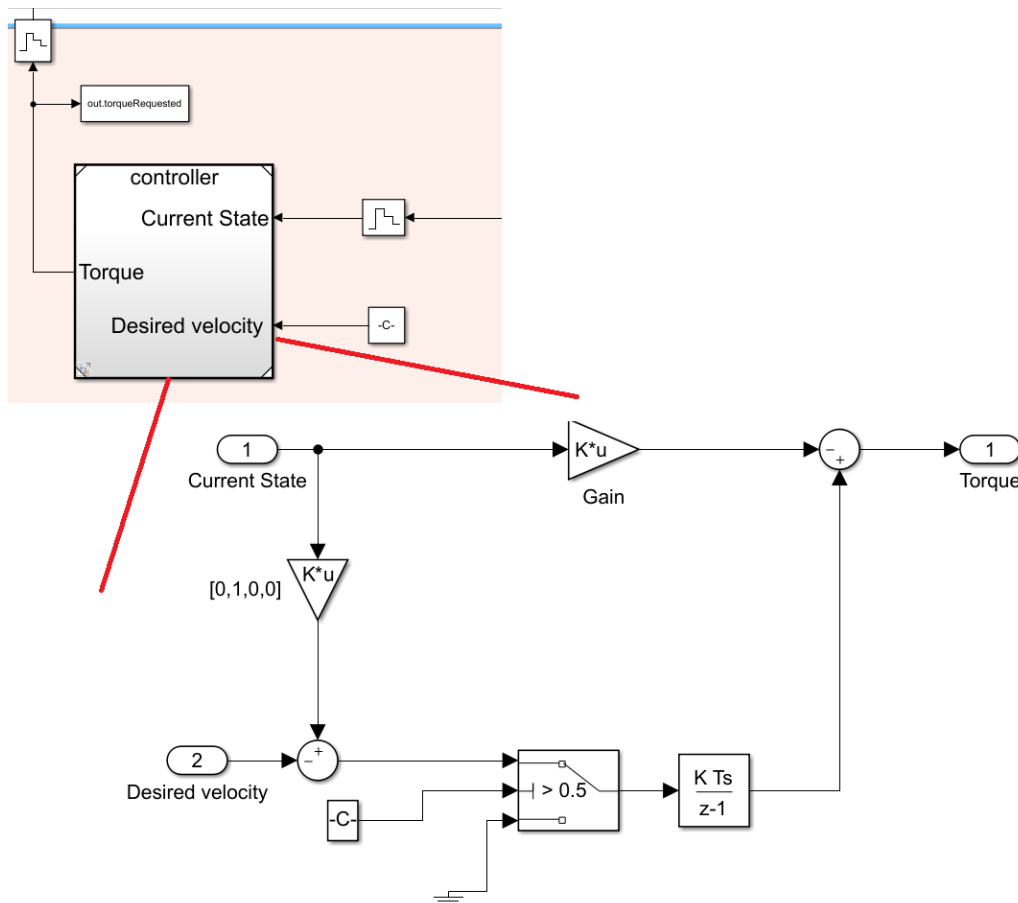


Figure 4.11: Modello Simulink del controllore discreto (vista ad alto livello e vista interna)

Per le stesse ragioni pratiche anche il modello del motore discreto (figura 4.12) ha subito una conversione: il controllore dell'anello di retroazione in corrente sarà un software eseguito ciclicamente su di un calcolatore e va dunque implementato a tempo discreto. Ricordando che la funzione che la *f.d.t.* del controllore *PI* a tempo continuo era:

$$\frac{0.001216s + 0.9965}{0.00122s}$$

e applicando il metodo di Tustin da tempo continuo a discreto con un $T_s = 0.001$ si ottiene che:

$$\frac{1.037z - 0.9557}{z - 1}$$

Quanto ottenuto a livello numerico qui sopra, è facilmente ottenibili tramite comandi per la modellizzazione dei sistemi dinamici che Matlab mette a disposizione: nello specifico, come si vede nel listato seguente, si tratta dei comandi *tf* e *c2d* (*continuous to discrete*):

```
10 PI_I = K_gi * tf([m.te,1],[m.te, 0])
11 PI_I_d = c2d(PI_I,Ts,'tustin')
```

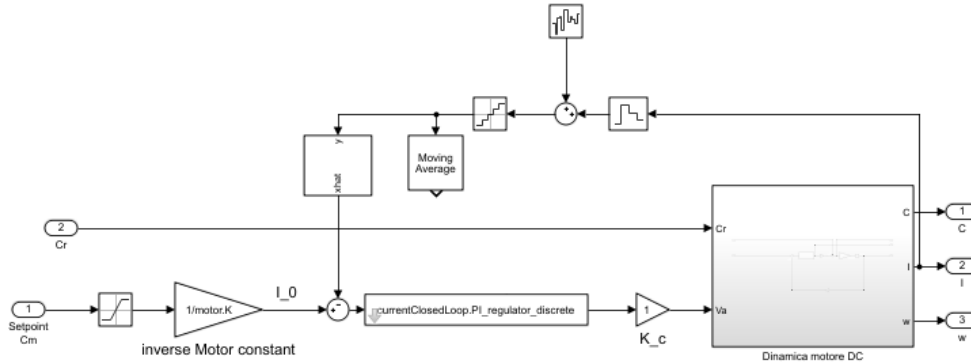


Figure 4.12: Modello Simulink del motore discreto

4.6 Sensori

A bordo del sistema sono presenti diversi sensori che permettono di definire e caratterizzare lo stato e il comportamento del veicolo auto bilanciato. In particolare abbiamo:

- **Encoder incrementale:** si occupa di misurare la rotazione della singola ruota;
- **I.M.U.:** Inertial Measasurement Unit, stima l'inclinazione dello chassis;
- **Sensore di corrente:** permette di rilevare la corrente che fluisce all'interno del motore, per così poter realizzare un controllo in corrente dello stesso;

Questa parte di sensoristica è stata presa in considerazione con l'obiettivo di rendere la simulazione più raffinata e accurata possibile: sono stati quindi inseriti all'interno dei file Simulink anche dei modelli che simulano la presenza dei sensori stessi.

Questo è necessario ed importante poiché nella realtà non esiste un modello matematico che genera i dati che poi sono dati in pasto al controllore ma si devono invece usare dei sensori che danno una stima dello stato del sistema. Un sensore presenta due principali caratteristiche:

- **Rumore:** è stato modellizzato come un *Band-Limited White Noise* con *Noise power* = [0.00000001] e *Sample time* = 0.001 tramite lo specifico blocco Simulink;

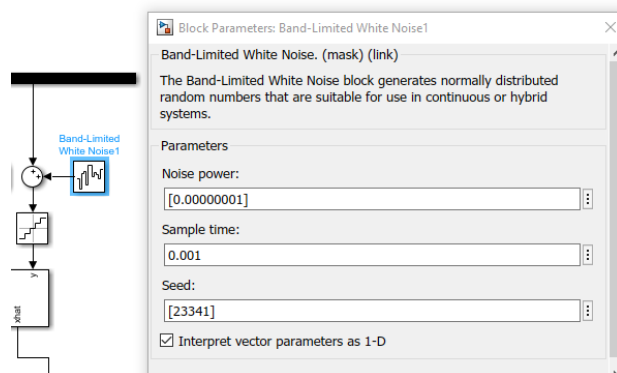


Figure 4.13: Blocco Simulink responsabile della generazione del rumore

- **Quantizzazione:** la lettura dei dati da un sensore, oltre ad avvenire ad un intervallo di tempo minimo e regolare, mostra anche un errore dovuto alla limitata sensibilità del sensore stesso o dell' *ADC* che effettua la lettura. Il blocco simulink *Quantizer* ha l'esatto scopo di simulare questo comportamento presente nei sensori reali.

Per come si presenta il sistema, sia l'*encoder* incrementale sia la *I.M.U.* restituiscono rispettivamente una stima di ϕ e θ . Per modellizzare quanto detto finora riguardante i sensori ed in particolare per l'*encoder* e l'*I.M.U.* è stato approntato il seguente blocco simulink:

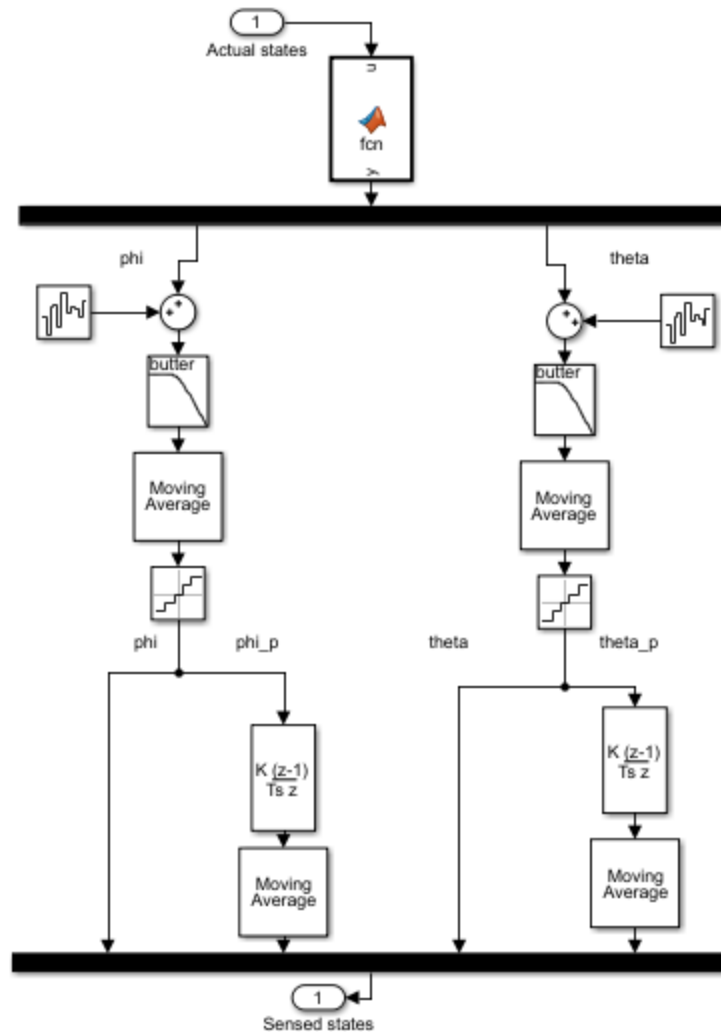


Figure 4.14: Modello Simulink dei sensori *encoder* e *I.M.U.* con Moving Average

Si può notare come in Fig.4.14 siano presenti due blocchi *Moving average* per ciascun sensore. Questo si è reso necessario per due ragioni:

- presentando il sistema del rumore, era necessario rimuoverlo attraverso un filtro digitale;
- la derivazione a tempo discreto del segnale presenta molti picchi e molti valori nulli; E' quindi necessario filtrare questi valori prima di passarli al controllore per evitare picchi di coppie troppo alti oppure nulli in brevi intervalli di tempo, rendendo necessario filtrare questi valori prima di passarli al controllore per evitare picchi di coppie troppo alti oppure nulli in brevi intervalli di tempo;

L'approccio con *Moving average* è stato quello inizialmente seguito: abbiamo poi però cercato anche di portare avanti un altro approccio in cui, la stima dello stato, venisse effettuata tramite filtro di Kalman. Il contesto infatti è quello in cui si ha un controllore che fornisce in uscita un comando in

coppia che cambia in base allo stato del sistema $(\phi, \dot{\phi}, \theta, \dot{\theta})$: i **problemi** che emergono sono però che:

- ϕ e θ vengono misurati ma i valori forniti dai sensori sono quantizzati (quindi sarà necessario attuare un filtraggio digitalmente);
- $\dot{\phi}$ e $\dot{\theta}$ non sono direttamente misurabili;

Una soluzione che si è soliti seguire in contesti del genere, è quella di andare appunto ad applicare un filtro di Kalman, come già specificato, per produrre così una stima dello stato.

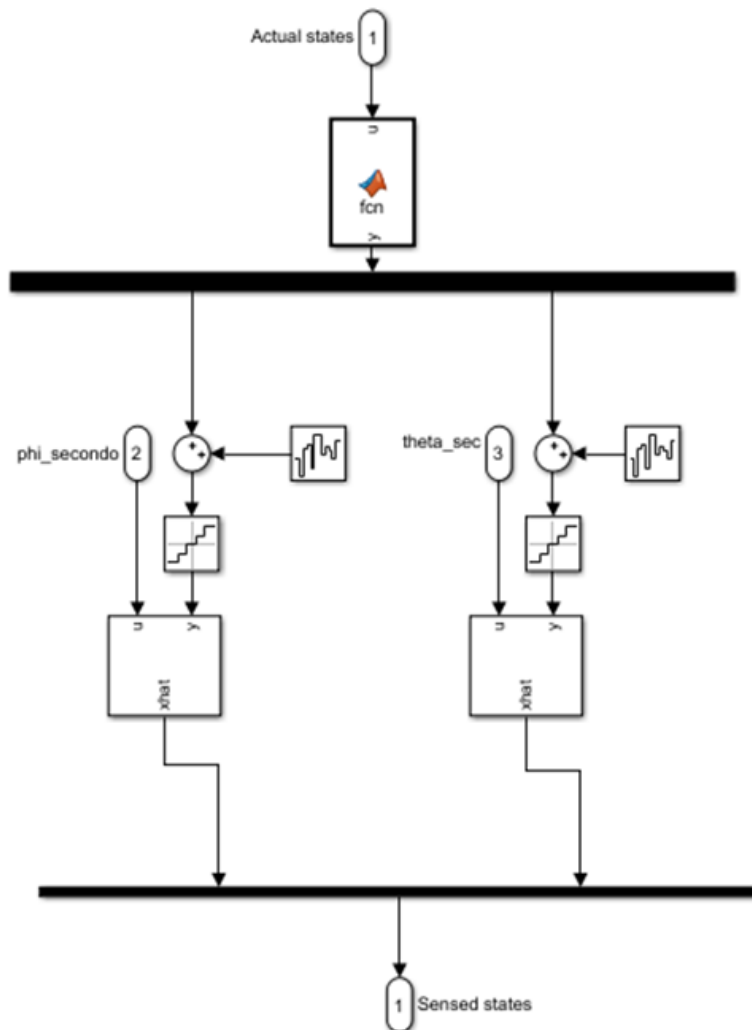


Figure 4.15: Modello Simulink dei sensori *encoder* e *I.M.U.* con Filtro di Kalman

Si presenta ora la necessità di modellizzare la presenza del sensore di corrente nel motore; per fare ciò si è modificato solamente l'anello di retroazione esterno che permette il controllo del motore in corrente:

Si può notare in figura 4.12 che il controllore è quello discreto presentato prima, mentre è stato inserito anche in questo caso un rumore sulla misura, una quantizzazione e un filtro sul segnale da passare al controllore.

Da sottolineare il fatto che, ognuno dei tre blocchi *Quantizer*, utilizzato per modellizzare altrettanti sensori, ha un valore diverso del parametro caratterizzante di *Quantization Interval*, dovuto al fatto che ognuno dei sensori ha sensibilità diverse:

- **Encoder incrementale:** presenta una sensibilità di $2^{11} \text{ bit} = 2048$ diverse combinazioni e letture possibili;

- **I.M.U.:** sensibilità del giroscopio di $131 \frac{s}{deg}$
- **Sensore di corrente:** il valore del sensore di corrente è un voltaggio letto da un Raspberry tramite ADC interno, il quale presenta sensibilità di 10 bit;

Appendix A

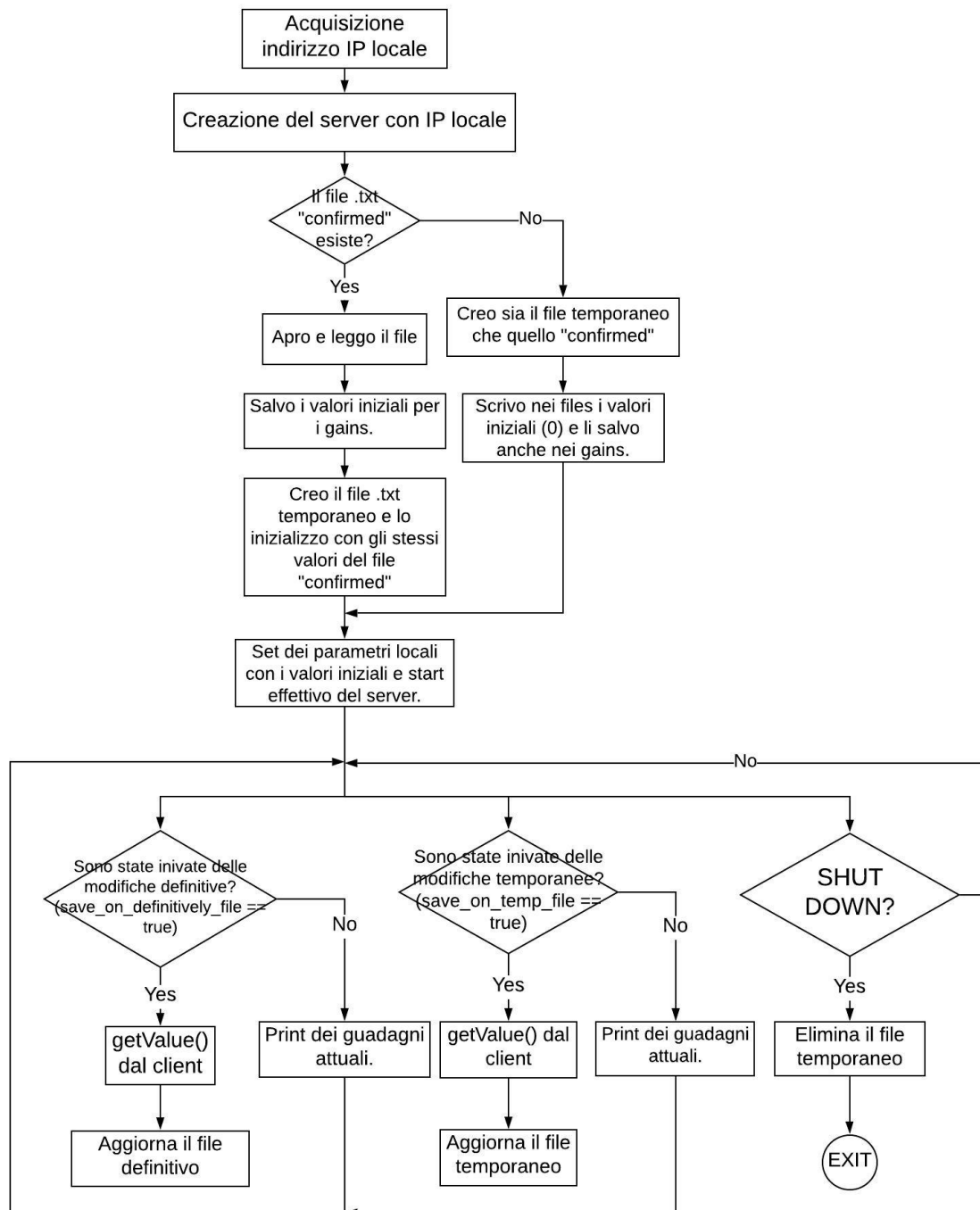


Figure A.1: Diagramma di flusso rappresentante le funzionalità del server

```

12 ##### LIBRARY IMPORT #####
13 from opcua import Server
14 from random import randint
15 from datetime import datetime
16 import os
17 import time
18 import socket
19 #####
20
21 ##### GET LOCAL MACHINE IP #####
22 s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
23 s.connect(("8.8.8.8", 80))
24 id = s.getsockname()[0]
25 s.close()
26 #####
27
28 ##### SET SERVER INSTANCE #####
29 server = Server()
30 url = "opc.tcp://" + id + ":4840"
31
32 server.set_endpoint(url)
33
34 name = "V.A.B. - Server Raspberry PI"
35 addspace = server.register_namespace(name)
36
37 node = server.get_objects_node()
38 Param = node.add_object(addspace, "Gain parameters")
39 #####
40
41 ##### FILE READING #####
42 fileName_temp = "GainParametersToController.txt"
43 fileName_confirmed = "GainParametersConfirmed.txt"
44
45 lineHeaders = ["K_phi ", "K_phi_p ", "K_theta ", "K_theta_p "]
46 initialValueGains = [0, 0, 0, 0]
47
48 if os.path.exists(fileName_confirmed):
49     print("Get initial values from file and create a temp one")
50     file_object_def = open(fileName_confirmed, "r+")
51     lines = file_object_def.readlines()
52     count = 0
53     # Strips the newline character
54     for line in lines:
55         print(line.strip())
56         values = line.split()
57         if count <= 3:
58             initialValueGains[count] = int(values[1])
59         else:
60             timestamp = str(values[0]) + str(values[1])
61             count = count + 1
62     file_object_def.close()
63
64     file_object_temp = open(fileName_temp, "w")
65     for i, header in zip(initialValueGains, lineHeaders):
66         print(header + str(i))
67         file_object_temp.write(header + str(i) + "\n")
68
69     # Time stamp from confirmed file
70     file_object_temp.write(timestamp)
71     file_object_temp.close()
72 else:
73     print("Creating a new files and set parameters to zero")
74     file_object_temp = open(fileName_temp, "w")
75     file_object_def = open(fileName_confirmed, "w")
76

```



```

77     for i, header in zip(range(4), lineHeaders):
78         print(header + "0")
79         file_object_temp.write(header + "0\n")
80         file_object_def.write(header + "0\n")
81
82     # current date and time
83     now = datetime.now()
84     timestamp = datetime.timestamp(now)
85     dt_object = datetime.fromtimestamp(timestamp)
86     file_object_temp.write(str(dt_object))
87     file_object_def.write(str(dt_object))
88
89     file_object_temp.close()
90     file_object_def.close()
91     #####
92
93     ##### PARAMETERS INIT #####
94     K_phi = Param.add_variable(addspace, "K_phi", initialValueGains[0])
95     K_phi_p = Param.add_variable(addspace, "K_phi_p", initialValueGains[1])
96     K_theta = Param.add_variable(addspace, "K_theta", initialValueGains[2])
97     K_theta_p = Param.add_variable(addspace, "K_theta_p", initialValueGains[3])
98     submit_to_controller = Param.add_variable(addspace, "Submit change to controller",
99         False)
100     submit_to_file = Param.add_variable(addspace, "Store definitively in file", False)
101     shut_down = Param.add_variable(addspace, "SHUT DOWN SERVER", False)
102
103     K_phi.set_writable()
104     K_phi_p.set_writable()
105     K_theta.set_writable()
106     K_theta_p.set_writable()
107     submit_to_controller.set_writable()
108     submit_to_file.set_writable()
109     shut_down.set_writable()
110     #####
111
112     server.start()
113
114     print("Server started at {}".format(url))
115
116     K1 = K_phi.get_value()
117     K2 = K_phi_p.get_value()
118     K3 = K_theta.get_value()
119     K4 = K_theta_p.get_value()
120
121     while True:
122         save_on_temp_file = submit_to_controller.get_value()
123         save_on_definitively_file = submit_to_file.get_value()
124         exit = shut_down.get_value()
125
126         if save_on_definitively_file:
127             # Write on file that store gains considered stable
128             K1 = K_phi.get_value()
129             K2 = K_phi_p.get_value()
130             K3 = K_theta.get_value()
131             K4 = K_theta_p.get_value()
132
133             gains = [K1, K2, K3, K4]
134             file_object_conf = open(fileName_confirmed, "w")
135             for header, gain in zip(lineHeaders, gains):
136                 file_object_conf.write(header + str(gain) + "\n")
137
138             # current date and time
139             now = datetime.now()
140
141             timestamp = datetime.timestamp(now)
142             dt_object = datetime.fromtimestamp(timestamp)

```

```

142     file_object_conf.write(str(dt_object))
143     file_object_conf.close()
144
145     submit_to_file.set_value(False)
146     print("Confirmed data")
147
148     if save_on_temp_file:
149         # Write on file that is read from controller on Raspberry
150         K1 = K_phi.get_value()
151         K2 = K_phi_p.get_value()
152         K3 = K_theta.get_value()
153         K4 = K_theta_p.get_value()
154
155         gains = [K1, K2, K3, K4]
156         file_object_temp = open(fileName_temp, "w")
157         for header, gain in zip(lineHeaders, gains):
158             file_object_temp.write(header + str(gain) + "\n")
159
160         # current date and time
161         now = datetime.now()
162
163         timestamp = datetime.timestamp(now)
164         dt_object = datetime.fromtimestamp(timestamp)
165         file_object_temp.write(str(dt_object))
166         file_object_temp.close()
167
168         submit_to_controller.set_value(False)
169         print("Submitted data to temp file that will be read from raspberry")
170
171     if not save_on_temp_file and not save_on_definitively_file:
172         # No change submitted
173         K = [K1, K2, K3, K4]
174         print(K)
175         time.sleep(2)
176
177     if exit:
178         print("Shut down server...")
179         break
180
181     if os.path.exists(fileName_temp):
182         print("Removing temp file...")
183         os.remove(fileName_temp)
184     else:
185         print("The temp file does not exist")
186     print("Server stopped")

```

Appendix B

Di seguito riportiamo il codice implementato per la gestione del controllo real time: alcune funzionalità, come la lettura degli ingressi e dei sensori, sono state sostituite da funzioni simulate.

```
187 #include <stdio.h>
188 #include <signal.h>
189 #include <unistd.h>
190 #include <alchemy/task.h>
191 #include <stdlib.h>
192 #include <string.h>
193 #include <strings.h>
194
195 #define MAX_LINE_LENGTH 100
196 #define WINDOW_WIDTH 200
197
198 #define TEMP_FILE_NAME "/home/Laboratorio_SistemiMeccatroniciII/functions/
    GainParametersToController.txt"
199 #define CONFIRMED_FILE_NAME "/home/Laboratorio_SistemiMeccatroniciII/functions/
    GainParametersConfirmed.txt"
200 RT_TASK hello_task;
201 double calcola_m_a(double*);
202
203 double k_simulazione_discreta[4];
204 double phi = 0;
205 double phi_precedente = 0;
206 double phi_p = 0;
207 double theta = 0;
208 double theta_precedente = 0;
209 double theta_p = 0;
210 double Ts = 0.001;
211 double phi_p_setpoint = 10;
212
213 double m_a_phi[WINDOW_WIDTH];
214 double m_a_phi_p[WINDOW_WIDTH];
215 double m_a_theta[WINDOW_WIDTH];
216 double m_a_theta_p[WINDOW_WIDTH];
217 double m_a_motore [WINDOW_WIDTH];
218
219 double integrator_controller = 0;
220 double integrator_controller_motor = 0;
221 double K_integrazione = 0.002;
222 double desired_cm = 0;
223 double actual_cm = 0;
224
225 void read_k(){
226     // Streams for file reading
227     FILE* tempGainFile;
228     FILE* confirmedGainFile;
229     char* line = (char*) malloc(sizeof(char) * (MAX_LINE_LENGTH + 1));
230
231     // First: try to open temp file (which temporary gains passed from opc ua client)
232     tempGainFile = fopen(TEMP_FILE_NAME, "r");
233     if(tempGainFile) {
234         // Read temporary parameters
```

```

235     printf("tempGainFile is open\n");
236     int index = 0;
237
238     while(fgets(line, (MAX_LINE_LENGTH + 1), tempGainFile) != NULL) {
239         if(index <= 3) {
240             char* token = strtok(line, " ");
241             token = strtok(NULL, line); // Get the second token (the double value)
242
243             printf("Gain value read: -> K%d = %s", index, token);
244             k_simulazione_discreta[index] = atof(token);
245             printf("Gain value parsed: -> K%d = %f\n", index, k_simulazione_discreta[
index]);
246         }
247         index++;
248     }
249     fclose(tempGainFile);
250     free(line);
251 } else {
252     // If there's no temporary passing file, try to open confirmed parameters
253     confirmedGainFile = fopen(CONFIRMED_FILE_NAME, "r");
254     if (confirmedGainFile) {
255         // Read confirmed parameters
256         printf("confirmedGainFile is open\n");
257         int index = 0;
258
259         while(fgets(line, (MAX_LINE_LENGTH + 1), confirmedGainFile) != NULL) {
260             if(index <= 3) {
261                 char* token = strtok(line, " ");
262                 token = strtok(NULL, line); // Get the second token (the double value)
263
264                 printf("Gain value read: -> K%d = %s", index, token);
265                 k_simulazione_discreta[index] = atof(token);
266                 printf("Gain value parsed: -> K%d = %f\n", index, k_simulazione_discreta[
index]);
267             }
268             index++;
269         }
270         fclose(confirmedGainFile);
271         free(line);
272     } else {
273         printf("Error on parameters reading\n");
274     }
275 }
276 }
277
278 void controllore(){
279
280     double cm_state_feedback = k_simulazione_discreta[0]*phi + k_simulazione_discreta
[1]*phi_p + k_simulazione_discreta[2]*theta + k_simulazione_discreta[3]*theta_p;
281     double difference_phi_p = phi_p - phi_p_setpoint;
282     integrator_controller = integrator_controller + K_integrazione*Ts*difference_phi_p;
283     desired_cm = integrator_controller - cm_state_feedback;
284 }
285
286 void controllore_motore(int counter){
287     //da verificare la media mobile, c'era un errore in matlab
288     //
289     double cm_setpoint = desired_cm;
290     double corrente_attuale_noise = rand();
291     //da rivedere le costanti
292     //
293     double k_t = 0.1;
294     double Ts = 0.0001;
295     double P = 1;
296     double I = 980;
297     if(cm_setpoint > 2)

```

```

296     cm_setpoint = 2;
297     if(cm_setpoint < -2)
298         cm_setpoint = -2;
299     double corrente_setpoint = cm_setpoint/k_t;
300     m_a_motore[counter] = corrente_attuale_noise;
301     double media_pesata = calcola_m_a(m_a_motore);
302     double corrente_filtrata = media_pesata;
303     double differenza_corrente = corrente_setpoint - corrente_filtrata;
304     integrator_controller_motor = I*Ts*differenza_corrente +
        integrator_controller_motor;
305     actual_cm = P*differenza_corrente + integrator_controller_motor;
306 }
307
308 void read_sensor_phi(int counter){
309     m_a_phi[counter] = rand();
310     double media_pesata = calcola_m_a(m_a_phi);
311     phi_precedente = phi;
312     phi = media_pesata;
313 }
314
315 void read_sensor_theta(int counter){
316     m_a_theta[counter] = rand();
317     double media_pesata = calcola_m_a(m_a_theta);
318     theta_precedente = theta;
319     theta = media_pesata;
320 }
321
322 void calculate_phi_p(int counter){
323     m_a_phi_p[counter] = (phi-phi_precedente)/Ts;
324     double media_pesata = calcola_m_a(m_a_phi_p);
325     phi_p = media_pesata;
326 }
327
328 void calculate_theta_p(int counter){
329     m_a_theta_p[counter] = (theta-theta_precedente)/Ts;
330     double media_pesata = calcola_m_a(m_a_theta_p);
331     theta_p = media_pesata;
332 }
333
334 double calcola_m_a(double* media_mobile){
335
336     double media_pesata = 0;
337     for(int i = 0; i<WINDOW_WIDTH;i++){
338         media_pesata = media_pesata + 1/WINDOW_WIDTH * media_mobile[i];
339     }
340     return media_pesata;
341 }
342 }
343
344 void wrap() {
345     printf("START");
346     int counter = 0;
347     RTIME period = 1000000000;
348     rt_task_set_periodic(NULL, TM_NOW, period);
349     RTIME now = rt_timer_read();
350     RTIME before = now - period;
351     while(1){
352         now = rt_timer_read();
353         Ts = (now - before)/1000000000;
354         read_sensor_phi(counter);
355         read_sensor_theta(counter);
356         calculate_phi_p(counter);
357         calculate_theta_p(counter);
358         read_k();
359         controlllore();
360         controlllore_motore(counter);

```

```
361     counter = (counter+1)%WINDOW_WIDTH;
362     printf("Actual motor torque: %f\n", actual_cm);
363     printf("Elapsed time: %llu\n",Ts);
364     before= now;
365 }
366
367     return;
368 }
369
370
371 int main(int argc, char* argv[])
372 {
373     char str[10] ;
374     printf("start task\n");
375     sprintf(str,"hello");
376
377     /* Create task
378      * Arguments: &task,
379      *             name,
380      *             stack size (0=default),
381      *             priority,
382      *             mode (FPU, start suspended, ...)
383      */
384     rt_task_create(&hello_task, str, 0, 50, 0);
385
386     /* Start task
387      * Arguments: &task,
388      *             task function,
389      *             function argument
390      */
391     rt_task_start(&hello_task, &wrap, 0);
392     pause();
393 }
```

Bibliography

- [1] *Controllo tramite retroazione dello stato, Controlli automatici, Fabio Previdi* https://cal.unibg.it/wp-content/uploads/controlli_automatici/Lez06.pdf
- [2] *RTOS Xenomai* <http://www.cs.ru.nl/lab/xenomai/>