Esercitazioni

Controllo Robusto e Nonlineare

Filippo Ottaviani 363509

A.A. 24/25

Indice

1 Soluzione ARE

Soluzione equazione di Riccati algebrica numerica.

1.1 Introduzione

L'**equazione algebrica di Riccati** (ARE) è un'equazione matriciale non lineare che rappresenta una soluzione del problema di controllo ottimo LQR (Linear Quadratic Regulator) ad orizzonte infinito. In LQR, l'ARE permette di calcolare la matrice di guadagno ottimale che minimizza un costo quadratico, garantendo prestazioni ottimali del sistema controllato.

1.2 Problema Linear Quadratic Regulator

Il **Problema Linear Quadratic Regulator** è una particolare casistica del problema di controllo ottimo che prevede alcune ipotesi di interesse pratico:

- la dinamica lineare.
- il costo quadratico, si vuole minimizzare, quindi, l'energia associata allo stato e all'input.
- nessun vincolo generico.

Sotto tali ipotesi si ottiene la seguente formulazione del problema:

$$\min_{(x,u)\in X\times U} \int_0^{t_f} (x(s)^T \boldsymbol{Q} x(s) + u(s)^T \boldsymbol{R} u(s))\, ds + x(t_f)^T \boldsymbol{S} x(t_f) \tag{1}$$

$$\text{s.t.} \begin{cases} \dot{x}(t) = \boldsymbol{A}x(t) + \boldsymbol{B}u(t) \quad \forall t \in [0, t_f] \\ x(0) = x_0 \end{cases} \tag{2}$$

dove si introducono le seguenti matrici di costo:

- $\boldsymbol{Q}$ rappresenta costo sullo stato; è simmetrica e semidefinita positiva, altrimenti esisterebbero stati pozzo in cui il sistema rimarrebbe intrappolato. Grazie alla sua simmetria può essere scritto come $Q = E^T E$.
- $\boldsymbol{R}$ rappresenta costo sul controllo; è simmetrica e definita positiva altrimenti il controllo avrebbe costo negativo o nullo e sarebbe impossibile minimizzare, esisterebbero infinite soluzioni.
- $\boldsymbol{S}$ rappresenta costo terminale; è simmetrica e semidefinita positiva. Ho omesso questa matrice di costo nella sezione relativa alla ARE poiché nei problemi ad orizzonte infinito non è contemplato lo stato finale.

La simmetria è richiesta perché implica la diagonalizzabilità e lo spettro reale. Il problema LQR ha due varianti principali:

- **Orizzonte finito**: la soluzione è un controllo $u^* : [0, t_f] \to \mathbb{R}^n$ che porta il sistema in uno stato finale $x(t_f)$.
- **Orizzonte infinito**: la soluzione è un controllo $u^* : [0, \infty] \to \mathbb{R}^n$ che minimizza i costi per $t \to \infty$.

1.3 LQR ad orizzonte finito

La soluzione del problema LQR ad orizzonte finito è il controllo risultante dalla minimizzazione 1.2 che posso scrivere come:

$$u^* = -\frac{1}{2}\boldsymbol{R}^{-1}\boldsymbol{B}^T\frac{\partial}{\partial x}V(t,x) \tag{3}$$

che posso sostituire nell'equazione di Hamilton-Jacobi-Bellman e, supponendo che la soluzione sia del tipo $V(t,x) = x^T P(t) x$, ottengo:

$$\begin{cases} -\dot{\boldsymbol{P}} = \boldsymbol{A}^T\boldsymbol{P} + \boldsymbol{P}\boldsymbol{A} + \boldsymbol{Q} - \boldsymbol{P}\boldsymbol{B}\boldsymbol{R}^{-1}\boldsymbol{B}^T\boldsymbol{P} \\ \boldsymbol{P}(t_f) = \boldsymbol{S} \end{cases} \tag{4}$$

nota come equazione differenziale di Riccati (DRE), soluzione del problema LQR ad orizzonte finito. Allora posso scrivere la soluzione a ciclo chiusa del sistema come segue:

$$u^* = -\frac{1}{2}\boldsymbol{R}^{-1}\boldsymbol{B}^T\boldsymbol{P}(t)x \tag{5}$$

1.4 LQR ad orizzonte infinito

Il problema LQR ad orizzonte infinito può essere risolto in maniera simile tenendo conto del fatto che, se l'istante finale t_f è arbitrariamente grande, il valore della V^* non dipenderà dal tempo. Si ottiene, quindi, l'equazione di HJB stazionaria che impone l'indipendenza di P dal tempo. Di conseguenza posso scrivere il controllo ottimo come:

$$\underline{u}^* = -\frac{1}{2}\boldsymbol{R}^{-1}\boldsymbol{B}^T\boldsymbol{P}\underline{x} \tag{6}$$

la matrice P può essere ricavata dall'equazione:

$$0 = \boldsymbol{A}^T\boldsymbol{P} + \boldsymbol{P}\boldsymbol{A} + \boldsymbol{Q} - \boldsymbol{P}\boldsymbol{B}\boldsymbol{R}^{-1}\boldsymbol{B}^T\boldsymbol{P} \tag{7}$$

nota come l'equazione algebrica di Riccati (ARE). Da notare che la ARE rappresenta la soluzione stazionaria della DRE, cioè per $\dot{P} = 0$

1.5 Calcolo della DRE

Per risolvere numericamente l'equazione algebrica di Riccati si può passare attraverso la formulazione dell'equazione differenziale di Riccati cercandone la soluzione stazionaria. Per risolvere la DRE occorre verificare tre condizioni:

1. positiva semidefinitezza di Q.
2. positiva definitezza di R.
3. controllabilità della coppia (A, B).

Per le prime due condizioni è sufficiente controllare il segno del più piccolo autovalore di Q e di R. Per la terza, invece, occorre costruire la matrice di controllabilità della coppia (A, B), $\mathcal{R}_{AB}$, e verificare se è a rango pieno. A queste ipotesi se ne aggiunge una quarta, ossia che la coppia (E, A) sia osservabile. Quest'ultima verifica assicura che la soluzione P_∞ della ARE è definita positiva e quindi il sistema controllato è asintoticamente stabile.

Verificate queste ipotesi, procedo con il calcolo di P_∞ che può essere ricavata dalla **stazionarizzazione della DRE** invece di risolvere direttamente l'equazione algebrica, si simula l'evoluzione dinamica della matrice $P(t)$. Se il sistema è stabilizzabile e rilevabile, partendo da una condizione iniziale $P(t_f) = P_f$, la matrice $P(t)$ convergerà a un valore costante integrando all'indietro.

1.6 Stazionarizzazione della DRE

Scrivo l'equazione differenziale come segue:

$$-\dot{P}(t) = A^T P(t) + P(t)A - P(t)BR^{-1}B^T P(t) + Q \tag{8}$$

e procedo con i seguenti passaggi:

1. Si imposta una condizione iniziale ($P(0) = 0$).
2. Si integra all'indietro nel tempo l'equazione differenziale numericamente.
3. Quando $\dot{P}(t) \approx 0$ (la derivata si annulla), il valore di $P(t)$ è la soluzione della ARE.

1.7 Codice

Lo script principale definisce il sistema, verifica le condizioni per il calcolo della ARE e invoca la funzione `solve_ARE_through_DRE.m`.

```
clear all;
clc;
```

```
close all;

addpath('funzioni','1')
pkg load control

% Definisco il sistema
A = [-1 2; -3 -4];
B = [1; 0];
E = eye(2);
Q = E'*E;
R = 1;

% Scelgo un orizzonte temporale T sufficientemente lungo
T_horizon = 100;

% Q deve essere semidefinita positiva
if ~is_pos_semidefinite(Q)
    error('ARE non risolvibile: Q non    semidefinita positiva!');
end

% R deve essere definita positiva
if ~is_pos_definite(R)
    error('ARE non risolvibile: R non    definita positiva!');
end

% (A,B) deve essere controllabile
if ~is_controllable(A, B)
    error('ARE non risolvibile: la coppia (A, B) non    controllabile!
       ');
end

% (E,A) deve essere osservabile
if ~is_observable(E, A)
    error('ARE non risolvibile: la coppia (A, B) non    osservabile!')
       ;
```

```
end

% Soluzione della ARE
P_inf = solve_ARE_through_DRE(A, B, Q, R, T_horizon);

fprintf('\nSoluzione dell equazione algebrica di Riccati (ARE)\n')
disp(P_inf)
```

La funzione `solve_ARE_through_DRE.m` integra all'indietro il risultato della DRE e restituisce la P_∞ ottenuta dopo un tempo che si ritiene sufficiente per la convergenza. Per utilizzare `ode45`, è stato necessario operare una vettorizzazione della matrice $\boldsymbol{P}$. Nel codice, questo avviene tramite il comando P(:), che impila le colonne della matrice $n \times n$ in un vettore $n^2 \times 1$.

```
function P_inf = solve_ARE_through_DRE(A, B, Q, R, T)
    [n, ~] = size(A);

    % Imposto e vettorizzo la condizione finale P(T) = 0
    P_tf = zeros(n);
    P_tf_vec = P_tf(:);

    % Definisco l'intervallo di integrazione all'indietro: [T, 0]
    tspan = [T, 0];

    % Integro all'indietro
    % ode45 chiama la funzione 'compute_DRE' ad ogni passo
    [t_backward, P_vec_backward] = ode45(@(t, P_vec) compute_DRE(P_vec
        , A, B, Q, R, n), tspan, P_tf_vec);

    % Riordino i risultati in avanti nel tempo (da t=0 a t=T)
    t = flipud(t_backward);
    P_vec = flipud(P_vec_backward);

    % Converto i risultati da forma vettoriale (usata da ode45)
    % a forma matriciale (un array 3D)
    num_points = length(t);
    P_sol = zeros(n, n, num_points);
    for i = 1:num_points
```

```
        P_vec_i = P_vec(i, :)';
        P_sol(:, :, i) = reshape(P_vec_i, n, n); % Ricostruisco la
            matrice n x n
    end

    % La soluzione dell'ARE    P(0)
    P_inf = P_sol(:,:,1);
end
```

La funzione `compute_DRE.m` calcola numericamente $\dot{P}$ attraverso la formulazione teorica della DRE. La funzione esegue l'operazione di *reshape* ad ogni passo di integrazione per calcolare la derivata matriciale, per poi ri-vettorizzare il risultato.

```
function Pdot_vec = compute_DRE(P_vec, A, B, Q, R, n)

    % Ricostruisco la matrice P (n x n) dal vettore P_vec (n^2 x 1)
    P = reshape(P_vec, n, n);

    % Calcolo la derivata P_dot usando l'equazione di Riccati
    Pdot = -(A'*P + P*A + Q - P*B*inv(R)*B'*P);

    % Vettorizzo l'output Pdot per restituirlo a ode45
    Pdot_vec = Pdot(:);
end
```

2 Regolazione del pendolo linearizzato

Regolazione con informazione parziale del carrello con pendolo linearizzato.

2.1 Introduzione

Il problema della regolazione del carrello con pendolo (*inverted pendulum on a cart*) con informazione parziale riguarda la **progettazione di un controllore** in grado di stabilizzare il sistema quando alcune variabili di stato non sono direttamente misurabili. Il sistema è descritto dalle equazioni dinamiche non lineari del pendolo inverso montato su un carrello. Linearizzando intorno alla posizione di equilibrio (pendolo verticale, carrello fermo) si ottiene il modello lineare. Lo stato è rappresentato da:

- Posizione del carrello x_c.
- Velocità del carrello v_c.
- Angolo del pendolo rispetto alla verticale θ_p.
- Velocità angolare del pendolo ω_p.

I parametri fondamentali per la caratterizzazione del sistema sono:

- Massa del carrello M.
- Massa del pendolo m.
- Lunghezza del pendolo l.
- Accelerazione gravitazionale g.

Le uscite sono l'unica informazione a disposizione del regolatore e si tratta di:

- Posizione del carrello x_c.
- Angolo del pendolo θ_p.

Scrivo il vettore di stato come segue:

$$x = \begin{bmatrix} x_c \\ v_c \\ \theta_p \\ \omega_p \end{bmatrix} = \begin{bmatrix} x_c \\ \dot{x_c} \\ \theta_p \\ \dot{\theta}_p \end{bmatrix} \tag{9}$$

2.2 Modello del sistema

Il modello del sistema con pendolo e carrello è un problema non lineare ma in questa esercitazione tratterò il modello linearizzato e la sua regolazione al fine di equilibrare il pendolo fissato sul carrello $\theta_p = 0$. Date le premesse, scrivo le matrici di design del sistema lineare:

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & -\frac{mg}{M} & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & \frac{(M+m)g}{Ml} & 0 \end{bmatrix} \quad B = \begin{bmatrix} 0 \\ \frac{1}{M} \\ 0 \\ \frac{1}{Ml} \end{bmatrix} \quad C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{10}$$

Considero nulla la matrice D. Il modello permette di scrivere la legge di controllo in retroazione dallo stato $u = -K\hat{x}$. Non avendo a disposizione lo stato completo ma una versione parziale e rumorosa, occorre stimarlo. La legge di controllo diventa, quindi, la seguente:

$$u = -K\hat{x} \tag{11}$$

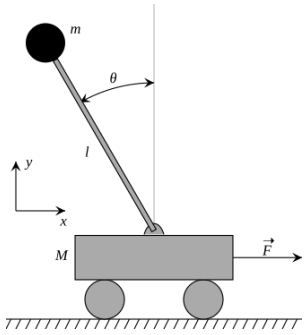

Figura 1: Pendolo inverso su un carrello

2.3 Principio di separazione

Il sistema di regolazione progettato si basa sul **Principio di separazione** quindi posso scomporre il problema in due sotto-problemi duali:

- Regolatore: progetto un guadagno K come se avessimo accesso a tutto lo stato x, minimizzando un indice di costo quadratico J; realizzo, quindi, un controllore LQR. Questo garantisce la stabilità e le prestazioni desiderate per la dinamica del sistema.
- Osservatore: progetto un osservatore con guadagno L per stimare lo stato $\hat{x}$ a partire dalle uscite misurate y. Questo oggetto, duale al precedente è chiamato Filtro di Kalman.

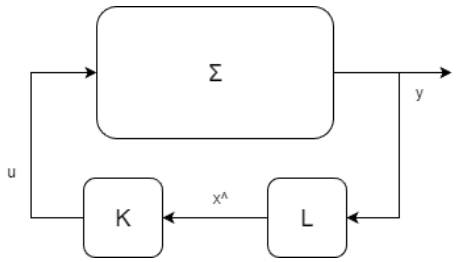

Figura 2: Sistema a ciclo chiuso con osservatore e regolatore

2.4 Condizioni preliminari

Prima di costruire il regolatore occorre verificare le seguenti tre ipotesi che abbiamo trattato come condizioni necessarie e sufficienti per poter realizzare il controllo:

- La coppia (A,B) deve essere **stabilizzabile** per rendere possibile il compito di stabilizzazione del sistema, cioè deve esistere un K per cui:

$$\sigma(A + BK) \subset \mathbb{C}^- \tag{12}$$

  Garantisce che posso stabilizzare il sistema con l'ingresso u.

- La coppia (A,C) deve essere **rilevabile** per accertare l'esistenza di un controllo stabilizzante dall'uscita per cui deve esistere L tale che:

$$\sigma(A - LC) \subset \mathbb{C}^- \tag{13}$$

  Garantisce che posso ricostruire lo stato (o almeno la sua parte instabile) dalle misure y. Senza questa proprietà, l'errore di stima non convergerebbe a zero e il controllo fallirebbe.

2.5 Costruzione del regolatore e dell'osservatore

2.5.1 Regolatore

Occorre, innanzitutto, definire le matrici di peso Q_{reg} e R_{reg}. Lo scopo del controllo è quello di penalizzare l'angolo del pendolo così da raggiungere il riferimento $\theta = 0$. Si intende dare una rilevanza molto minore al resto dello stato, in particolare in molti esempi online si lascia una penalità molto ridotta per la velocità che si lascia libera di variare pur di equilibrare il pendolo. Per quanto riguarda il controllo, non è di interesse penalizzarlo. Valori tipici in casi simili a questo sono:

$$Q_{reg} = \begin{bmatrix} 10 & 0 & 0 & 0 \\ 0 & 0.1 & 0 & 0 \\ 0 & 0 & 100 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad R_{reg} = 0.1 \tag{14}$$

A questo punto, posso calcolare il guadagno del regolatore sulla base delle matrici (A, B, Q_{reg}, R_{reg}) grazie alla funzione `lqr`, una funzione della libreria `control`. Ho deciso però di sfruttare i l risultato dell'esercitazione precedente, trovando P_∞ attraverso la ARE e calcolare, di conseguenza, il guadagno K come segue:

$$u = -\underbrace{R^{-1}B^T P_\infty}_{K} x \tag{15}$$

2.5.2 Osservatore

Il principio di separazione la costruzione dell'osservatore è duale a quella del regolatore con la differenza delle matrici Q ed R utilizzate. La prima, Q_{obs}, rappresenta la covarianza di processo mentre, la seconda, R_{obs} la covarianza di misura. Anche in questo caso, uso matrici di utilizzo comune negli esempi reperibili.

$$Q_{obs} = \begin{bmatrix} 10 & 0 & 0 & 0 \\ 0 & 0.1 & 0 & 0 \\ 0 & 0 & 100 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad R_{obs} = 0.1 \tag{16}$$

Per trovare il guadagno dell'osservatore L ho utilizzato la stesso procedimento descritto per il regolatore sulla base delle matrici (A^T, C^T, Q_{obs}, R_{obs}) e, grazie alla funzione realizzata nell'esercitazione precedente trovo la P_∞ da cui:

$$L = (R^{-1}CP_\infty)^T \tag{17}$$

2.6 Sistema a ciclo chiuso

A questo punto si simula il sistema a ciclo chiuso completo di osservatore e regolatore. Scrivo la dinamica del sistema con le nuove matrici complete dei guadagni K e L:

$$\begin{bmatrix} \dot{x} \\ \dot{\hat{x}} \end{bmatrix} = \begin{bmatrix} A & -BK \\ LC & A - BK - LC \end{bmatrix} \begin{bmatrix} x \\ \hat{x} \end{bmatrix} \tag{18}$$

Scelgo, quindi, una vettore di stato per le condizioni di stato $x_0 = [1, 0, 0.3, 0]^T$ e la sua versione stimata $\hat{x}_0 = [0, 0, 0, 0]^T$. A questo punto, simulo il sistema e grafico l'andamento di stato reale, stimato ed errore nel tempo.

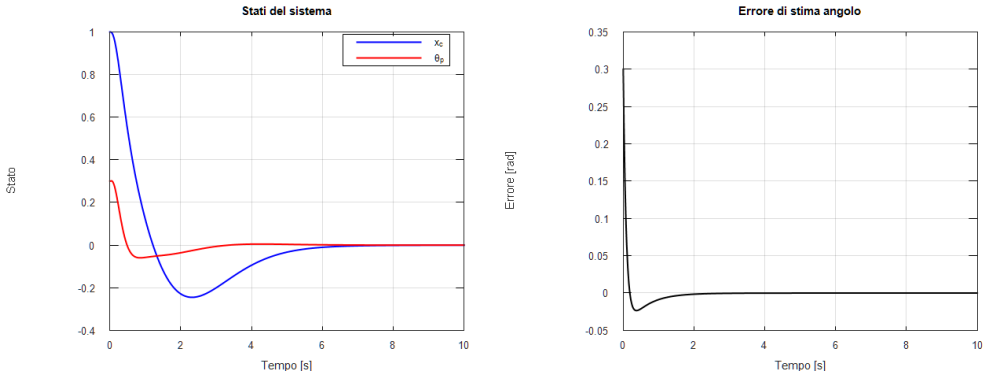

Figura 3: Andamento degli stati del sistema (sinistra) e dell'errore di stima del angolo del pendolo (destra)

2.7 Codice

Lo script principale `lin_pendulum_control` definisce i parametri fisici della scena e le matrici di design derivate dal modello, verifica le condizioni per attuare il controllo LQR e definisce le matrici di penalità per LQR (Q ed R). Calcola, dunque, i guadagni K ed L attraverso la funzione `solve_ARE_through_DRE` (ho inserito anche un confronto con quello ottenuto dalla funzione `lqr`) involca la funzione per la simulazione del sistema a ciclo chiuso.

```
clear;
clc;
close all;

pkg load control;

addpath('1', '2', 'funzioni')

% Parametri del sistema
M = 2.0;    % Massa del carrello [kg]
m = 0.1;    % Massa del pendolo [kg]
l = 0.5;    % Lunghezza del pendolo [m]
g = 9.81;   % Accelerazione di gravit  [m/s^2]

% Tempo per l'integrazione di P_inf
T_horizon = 10

% Definizione delle modello linearizzato
```

```
A = [0, 1, 0, 0;
     0, 0, -m*g/M, 0;
     0, 0, 0, 1;
     0, 0, (M+m)*g/(M*l), 0];

B = [0;
     1/M;
     0;
     1/(M*l)];

C = [1, 0, 0, 0;
      0, 0, 1, 0];

D = zeros(size(C,1), size(B,2));

% (A,B) deve essere controllabile
if is_controllable(A, B)
    disp('La coppia (A, B)    controllabile')
else
    error('ARE non risolvibile: la coppia (A, B) non    controllabile!
       ');
end

% (A, C) deve essere rilevabile
if is_detectable(A, C)
    disp('La coppia (A, C)    rilevabile')
else
    error('ARE non risolvibile: la coppia (A, C) non    rilevabile!');
end

% Progetto del regolatore LQR
Q_reg = diag([10, 1, 100, 1]); % Pesi sugli stati: [pos, v_pos, theta,
    v_theta]
R_reg = 0.1;                   % Peso sull'ingresso di controllo
```

```
K_fun = lqr(A, B, Q_reg, R_reg);
P_inf_reg = solve_ARE_through_DRE(A, B, Q_reg, R_reg, T_horizon);
K_cal = inv(R_reg) * B' * P_inf_reg;
disp('\nGuadagno del controllore K:\n');
disp('Guadagno K della funzione lqr');
disp(K_fun);
disp('Guadagno K calcolato dalla mia ARE');
disp(K_cal);
K = K_cal

% Progetto dell'osservatore dello stato
Q_obs = diag([100, 100, 100, 100]); % Rumore di processo fittizio
R_obs = diag([1, 1]);               % Rumore di misura fittizio (su
    pos e angolo)

% Calcolo il guadagno dell'osservatore per dualit
L_fun = lqr(A', C', Q_obs, R_obs)';
P_inf_obs = solve_ARE_through_DRE(A', C', Q_obs, R_obs, T_horizon);
L_cal = (inv(R_obs) * C * P_inf_obs)';
disp('Guadagno dell osservatore L:\n');
disp('Guadagno L della funzione lqr');
disp(L_fun);
disp('Guadagno L calcolato dalla mia ARE');
disp(L_cal);
L = L_cal

% Simulazione del sistema a ciclo chiuso
t_span = 0:0.01:10;

% Condizione iniziale
x0_sys = [1; 0; 0.3; 0];
x0_obs = [0; 0; 0; 0];
x0_total = [x0_sys; x0_obs];

% Definizione della dinamica completa
```

```
dynamics = @(t, z) closed_loop_dynamics(t, z, A, B, C, K, L);

[t, z] = ode45(dynamics, t_span, x0_total);

% Estrazione stati
x_real = z(:, 1:4); % Stati veri
x_hat = z(:, 5:8); % Stati stimati

% Plot dello stato
plot_state(x_real, x_hat, t)
```

La funzione `closed_loop_dynamics` calcola le matrici del sistema a ciclo chiuso includendo osservatore e regolatore e simula l'andamento dello stato e della sua stima.

```
function dz = closed_loop_dynamics(t, z, A, B, C, K, L)
    % Estraggo gli stati reali e stimati
    x = z(1:4);
    x_hat = z(5:8);

    % Uscita
    y = C * x;

    % Legge di controllo basata sullo stato stimato
    u = -K * x_hat;

    % Dinamica del sistema reale
    dx = A * x + B * u;

    % Dinamica dell'osservatore
    dx_hat = A * x_hat + B * u + L * (y - C * x_hat);

    % Combina derivate
    dz = [dx; dx_hat];
end
```

La funzione `plot_state` serve a mostrare i grafici degli andamenti dello stato della sua stima e dell'errore di stima.

```
function plot_state(x_real, x_hat, t)
```

```
    graphics_toolkit("gnuplot"); # fix per i grafici
    clf
    figure('Name', 'Regolazione dall uscita del pendolo linearizzato')
       ;

    subplot(1,2,1);
    plot(t, x_real(:,1), 'b', 'LineWidth', 2);
    hold on;
    plot(t, x_real(:,3), 'r', 'LineWidth', 2);
    title('Stati del sistema');
    xlabel('Tempo [s]');
    ylabel('Stato');
    legend('x_{c}', '\theta_p');
    grid on;

    subplot(1,2,2);
    plot(t, x_real(:,3) - x_hat(:,3), 'k', 'LineWidth', 2);
    title('Errore di stima angolo');
    xlabel('Tempo [s]');
    ylabel('Errore [rad]');
    grid on;
end
```

3 Soluzione equazioni nonlineari

Soluzione equazioni nonlineari analitica e numerica con commenti su esistenza, unicità e completezza delle soluzioni.

3.1 Introduzione

A differenza dei sistemi lineari, che rappresentano un'astrazione spesso ideale, i sistemi non lineari descrivono in modo più accurato la maggior parte dei fenomeni del mondo reale. Tuttavia, questa maggiore aderenza alla realtà introduce notevoli complessità. Una delle differenze fondamentali risiede nel comportamento delle loro soluzioni. Mentre per i sistemi lineari l'esistenza, l'unicità e la stabilità globale sono spesso garantite, per i sistemi non lineari è cruciale analizzare nel dettaglio le condizioni di:

- **Esistenza**: esiste almeno una soluzione che parte da una data condizione iniziale?
- **Unicità**: quella soluzione è l'unica per il sistema?
- **Completezza**: la soluzione è tale per ogni istante di tempo $t \geq 0$?

Una volta verificate queste condizioni è possibile cercare la soluzione del sistema attraverso uno tra i seguenti due metodi:

- **Analitico**: derivazione manuale dell'equazione differenziale ordinaria. Si tratta di una soluzione esatta e continua.
- **Numerico**: soluzione basata sulla funzione di risoluzione di equazioni differenziali ordinarie `ode45` utilizzata in Octave e MATLAB, basata su un metodo esplicito di Runge-Kutta. Si tratta di un approssimazione puntuale della soluzione.

3.2 Definizione del sistema

Per questa esercitazione ho scelto due sistemi non lineari semplici per poter svolgere i passaggi analitici:

$$\dot{x}_1 = x^2 \quad \text{e} \quad \dot{x}_2 = -x^3 \tag{19}$$

Dato un sistema non lineare autonomo del tipo:

$$\begin{cases} \dot{x} = f(t, x) \\ x(t_0) = x_0 \end{cases} \tag{20}$$

le condizioni di esistenza, unicità e completezza possono essere verificate senza il calcolo diretto dell'equazione differenziale non lineare che è spesso complesso o impossibile. In particolare, si sfruttano i seguenti tre teoremi fondamentali per i sistemi non lineari.

3.3 Esistenza

Teorema di Caratheodory: assumendo che $f(t,x)$ sia continua rispetto ad $x(t)$ e continua a tratti rispetto a t, allora esiste una soluzione $x(t)$ al sistema 20. L'esistenza dipende dalla continuità della $f(x)$ ed è verificata per entrambe le funzioni non lineari che definiscono la dinamica dei due sistemi.

3.4 Unicità

Lemma di Lipschitz per l'unicità: Sia $f : [a,b] \times D \subset \mathbb{R}^n \to \mathbb{R}^n$ continua su D $\forall t \in [a,b]$. Supponendo che $\frac{\partial f}{\partial x}$ esista e che sia continua su $[a,b] \times D$. Se, per un insieme convesso $W \subset D$, esiste una costante $L \geq 0$ tale che:

$$||\frac{\partial f}{\partial x}(t,x)|| \leq L \quad \forall (t,x) \in [a,b] \times W \tag{21}$$

allora la f è lipschitziana localmente. Questo implica l'unicità della soluzione in un intervallo limitato. Per comodità ho verificato una condizione più forte, la continua derivabilità:

$$\frac{\partial f_1}{\partial x} = 2x \quad \frac{\partial f_2}{\partial x} = -3x^2 \tag{22}$$

Entrambe le derivare sono continue dunque i sistemi sono localmente lipschitziani quindi, la soluzione è unica, almeno localmente. Non posso estendere globalmente l'unicità delle soluzioni perché questo richiederebbe la lipschitzianità globale delle due funzioni non lineari f_1 ed f_2. Questa condizione non è verificata poiché:

$$\lim_{t\to\infty} \frac{\partial f_1}{\partial x} = \lim_{t\to\infty} 2x = \infty \quad \text{e} \quad \lim_{t\to\infty} \frac{\partial f_2}{\partial x} = \lim_{t\to\infty} -3x^2 = -\infty \tag{23}$$

entrambe le funzioni hanno derivate illimitate quindi non sono globalmente lipschitziane.

3.5 Completezza

Teorema di Lipschitz per la completezza: sia $f(x,t)$ continua a tratti rispetto a t tale che soddisfa la condizione di Lipschitz in x $\forall t \geq t_0$ $\forall x \in D \subset R^n$. Sia $W \subset D$ un insieme compatto e $x_0 \in W$. Assumendo che qualsiasi soluzione a 20 appartiene a W. Allora, per tale sistema, esiste una soluzione unica definita $\forall t \geq t_0$. Occorre verificare, quindi. che la soluzione $x(t)$ rimanga in un insieme compatto. In questo caso, invece di calcolare la soluzione per valutarla si può fare una valutazione qualitativa del suo andamento. Nel caso del primo sistema, la dinamica dello stato è quadratica quindi sempre positiva, di conseguenza, la soluzione non è limitata per $t \to \infty$. Mi aspetto, quindi, che la soluzione assuma valori infiniti in tempo finito (*finite time escape*). Nel caso del secondo sistema, invece, per valori positivi di t la dinamica rimane negativa, quindi la soluzione risulta decrescente e tendente a 0. La condizione è soddisfatta, dunque il secondo sistema ammette una soluzione completa ($\forall t \geq 0$).

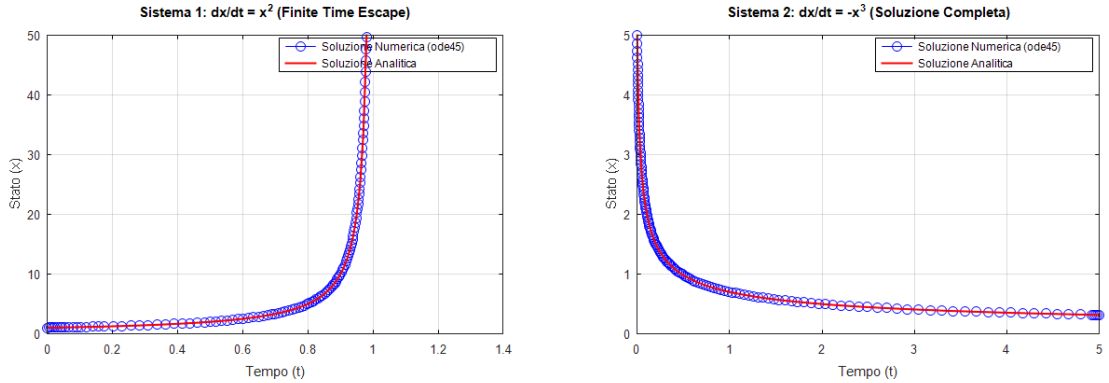

Figura 4: Soluzioni analitiche e numeriche del sistema 1 (sinistra) e del sistema 2 (destra)

3.6 Codice

Lo script `nonlinear_systems` definisce i due sistemi e invoca su ciascuno la funzione di soluzione analitica e quella di soluzione numerica.

```
clear all;
clc;
close all;

addpath('3')

%% SISTEMA 1: dx/dt = x^2
% Definisco il problema
f1 = @(t, x) x^2; % Funzione di stato
f1_analitica = @(t, x0) x0 ./ (1 - t.*x0); % Soluzione analitica
t_span1 = [0, 5];  % Intervallo di tempo desiderato
x0_1 = 1;          % Condizione iniziale

% Calcolo la soluzione numerica (ode45)
[t_num1, x_num1] = numeric_solution(f1, t_span1, x0_1);

% Calcolo la soluzione analitica (solo per il plotting, evitando l'
    asintoto)
t_plot_analitico1 = linspace(0, 0.99, 100);
```

```
[t_an1, x_an1] = analytical_solution(f1_analitica, t_plot_analitico1,
    x0_1);

% Plotting
titolo1 = 'Sistema 1: dx/dt = x^2 (Finite Time Escape)';
plot_sol(1, titolo1, t_num1, x_num1, t_an1, x_an1);
ylim([0, 50]); % Limito l'asse y per vedere il comportamento

%% SISTEMA 2: dx/dt = -x^3
% Definisco il problema
f2 = @(t, x) -x^3; % Funzione di stato
f2_analitica = @(t, x0) x0 ./ sqrt(1 + 2.*t.*x0.^2); % Soluzione
    analitica
t_span2 = [0, 5];  % Intervallo di tempo
x0_2 = 5;          % Condizione iniziale

% Calcolo la soluzione numerica (ode45)
[t_num2, x_num2] = numeric_solution(f2, t_span2, x0_2);

% Calcolo l'analitica sugli stessi punti della numerica per un
    confronto diretto
[t_an2, x_an2] = analytical_solution(f2_analitica, t_num2, x0_2);

% Plotting
titolo2 = 'Sistema 2: dx/dt = -x^3 (Soluzione Completa)';
plot_sol(2, titolo2, t_num2, x_num2, t_an2, x_an2);
```

La funzione `numeric_solution` utilizza `ode45` per derivare x.

```
function [t_num, x_num] = numeric_solution(fun_handle, t_span, x0)

    % Opzioni standard per il risolutore
    options = odeset('RelTol', 1e-6, 'AbsTol', 1e-9);
    disp('Avvio risolutore numerico ode45...');
    try
        % ode45 gestisce gli errori di "finite time escape"
        [t_num, x_num] = ode45(fun_handle, t_span, x0, options);
```

```
        disp('Soluzione numerica calcolata.');
    catch ME
        disp(['Errore durante ode45: ', ME.message]);
        disp('La simulazione potrebbe essersi interrotta.');
        t_num = [];
        x_num = [];
    end
end
```

La funzione `analytical_solution` calcola i valori a partire dalla formulazione ottenuta con il calcolo analitico svolto a mano.

```
function [t_an, x_an] = analytical_solution(fun_analitica_handle,
   t_punti, x0)
    % Calcolo della soluzione analitica
    disp('Calcolo soluzione analitica...');
    t_an = t_punti;
    x_an = fun_analitica_handle(t_an, x0);
    disp('Soluzione analitica calcolata.');
end
```

La funzione `plot_sol` genera un grafico di confronto tra soluzione numerica e analitica.

```
function plot_soluzioni(fig_num, titolo, t_num, x_num, t_an, x_an)
    graphics_toolkit("gnuplot"); # fix per i grafici

    % Grafico di confronto
    figure(fig_num);
    plot(t_num, x_num, 'b-o', 'LineWidth', 1.5, 'MarkerSize', 4); %
       Numerica
    hold on;
    plot(t_an, x_an, 'r-', 'LineWidth', 2); % Analitica
    title(titolo);
    xlabel('Tempo (t)');
    ylabel('Stato (x)');
    legend('Soluzione Numerica (ode45)', 'Soluzione Analitica');
    grid on;
    hold off;
end
```

4 Analisi diodo tunnel

Analisi degli equilibri circuito con diodo tunnel.

4.1 Introduzione

Il diodo tunnel è un componente elettronico in cui il semiconduttore viene pesantemente drogato per sfruttare l'effetto tunnel quantistico per operare a frequenze molto elevate, generando una caratteristica di resistenza negativa nella sua curva, ideale per oscillatori, amplificatori e circuiti a microonde ad alta velocità, nonostante le basse potenze, grazie alla sua rapidità di commutazione.

Per analizzare i punti di equilibrio occorre caratterizzare il componente attraverso un modello non lineare, trovare i valori che annullano la dinamica e caratterizzare la stabilità di ciascun attraverso lo jacobiano definendo se un punto di equilibrio è una sella, un nodo (stabile o instabile), un fuoco (stabile o instabile) o un centro.

4.2 Modellazione del dispositivo

Il modello più accreditato del diodo tunnel è definito dal sistema suggerito dal Politecnico di Milano:

$$\begin{cases} \dot{x}_1 = -\frac{R}{L}x_1 + \frac{1}{L}x_2 \\ \dot{x}_2 = -\frac{1}{C}x_1 + \frac{\alpha}{C}x_2 - \frac{\beta}{C}x_2^3 \end{cases} \tag{24}$$

tali che $v_L = L\dot{x}_1$ è la tensione sull'induttore e $i_C = C\dot{x}_2$ la corrente sul condensatore.

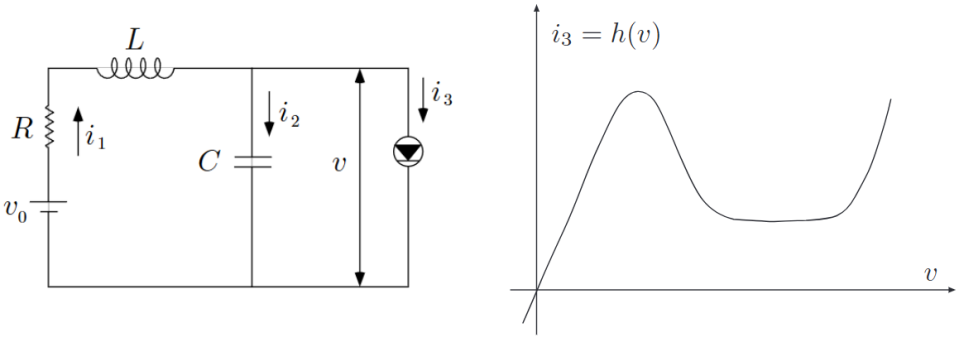

Figura 5: Circuito schematico del diodo tunnel (sinistra) e la sua curva caratteristica per la corrente che lo attraversa (destra).

4.3 Analisi locale dei sistemi non lineari

Il comportamento di un sistema non lineare intorno ad un punto di equilibrio è simile a quello di un sistema lineare. Per comprendere l'andamento locale del sistema non lineare, quindi, lo si linearizza e

se ne studia il comportamento attorno ai punti di equilibrio. Per fare ciò, ci si avvale dello sviluppo in serie di Taylor che, data l'evoluzione del sistema $x(t) = x_e + \delta(t)$, descrive l'andamento attorno ad x_e come segue:

$$f(x_e + \delta(t)) \approx f(x_e) + \left.\frac{\partial f}{\partial x}\right|_{x=x_e} \cdot \delta(t) \tag{25}$$

Per definizione $f(x_e) = 0$ mentre il termine restante non è altro che la matrice jacobiana $J(x_e)$ che contiene le derivate parziali della funzione $f(x)$ in x_e moltiplicata per $\delta(t)$. In questo modo ottengo un sistema lineare di cui posso studiare gli autovalori e, di conseguenza, il comportamento:

$$x(t) = J(x_e)x(t) \tag{26}$$

Per cercare i punti di equilibrio assegno dei valori arbitrari alle costanti ($R = 1$, $B = 1$, $\alpha = 3$ e $\beta = 1$), pongo $\dot{x}_1$ e $\dot{x}_2$ a 0 e risolvo il sistema di due equazioni a due incognite:

$$\begin{cases} -\frac{R}{L}x_1 + \frac{1}{L}x_2 = 0 \\ -\frac{1}{C}x_1 + \frac{\alpha}{C}x_2 - \frac{\beta}{C}x_2^3 = 0 \end{cases} \quad \Rightarrow \quad \begin{cases} x_1 = x_2 \\ 2x_2 - x_2^3 = 0 \end{cases} \tag{27}$$

Ne ricavo tre soluzioni che si traducono in punti di equilibrio $P_1 = (0, 0)$, $P_2 = (\sqrt{2}, \sqrt{2})$ e $P_3 = (-\sqrt{2}, -\sqrt{2})$.

4.4 Classificazione degli equilibri

Questo sistema approssima il comportamento del sistema non lineare originale in un intorno sufficientemente piccolo del punto di equilibrio x_e. Lo studio di questo sistema lineare è si basa sulla valutazione degli autovalori (λ_i) della matrice jacobiana $J(x_e)$:

$$J(x_e) = \left.\begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \dots \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \dots \\ \dots & \dots & \dots \end{bmatrix}\right|_{x=x_e} \tag{28}$$

Nella valutazione della natura (reale o immaginaria) e del segno dei suoi autovalori si distinguono tre casi fondamentali:

- **Autovalori reali**: allora il sistema può essere scritto come:

$$\begin{cases} \dot{z_1}(t) = \lambda_1 z_1(t) \\ \dot{z_2}(t) = \lambda_2 z_2(t) \end{cases} \tag{29}$$

  In base al valore degli autovalori si determina la natura del punto di equilibrio:

  – Se tutti gli autovalori sono negativi l'equilibrio è un *nodo stabile*.

- Se tutti gli autovalori sono positivi l'equilibrio è un *nodo instabile*.
- Se gli autovalori hanno segni discordi l'equilibrio è un punto di *sella*.

- **Autovalori complessi coniugati**: allora il sistema può essere scritto come:

$$\begin{cases} \dot{z_1}(t) = \alpha z_1(t) - \beta z_2(t) \\ \dot{z_2}(t) = \beta z_1(t) + \alpha z_2(t) \end{cases} \tag{30}$$

In base al valore degli autovalori si determina la natura del punto di equilibrio:

- Se il parametro α è negativo lo stato del sistema converge a 0, l'equilibrio è un *fuoco stabile*.
- Se il parametro α è positivo lo stato del sistema diverge, l'equilibrio è un *fuoco stabile*.
- Se il parametro α è nullo lo stato oscilla intorno a 0, l'equilibrio è un *centro*.

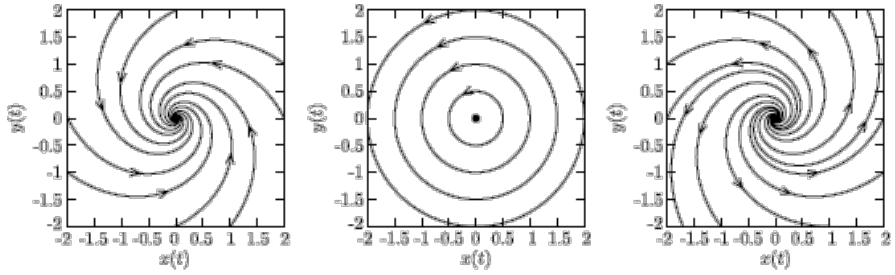

Figura 6: Visualizzazione planare di un fuoco stabile, un centro (centro) e un fuoco instabile (destra)

- **Autovalori nulli**: Se c'è almeno un autovalore nullo l'origine non è l'unico punto di equilibrio, ci sono infatti infiniti punti di equilibrio dato che $Ax = 0$ anche per $x \neq 0$.

Procedo con la **classificazione dei punti di equilibrio** tramite lo studio degli autovalori trovati. Grazie al calcolo simbolico trovo lo jacobiano e caratterizzo come segue i punti di equilibrio:

- Per il punto di equilibrio $(0, 0)$ gli autovalori sono $\lambda_1 = 2,73$ e $\lambda_2 = -0,73$. Il primo è positivo e, dunque, instabile mentre il secondo è negativo quindi stabile. Questo denota che l'origine è un punto di sella.
- Per il punto di equilibrio $(\sqrt{2}, \sqrt{2})$ gli autovalori sono $\lambda_1 = \lambda_2 = -2$. Essendo reali, negativi e coincidenti, si tratta di un nodo stabile (localmente asintoticamente stabile).
- Anche per il punto di equilibrio $(-\sqrt{2}, -\sqrt{2})$ gli autovalori sono $\lambda_1 = \lambda_2 = -2$ quindi ha le stesse proprietà del precedente.

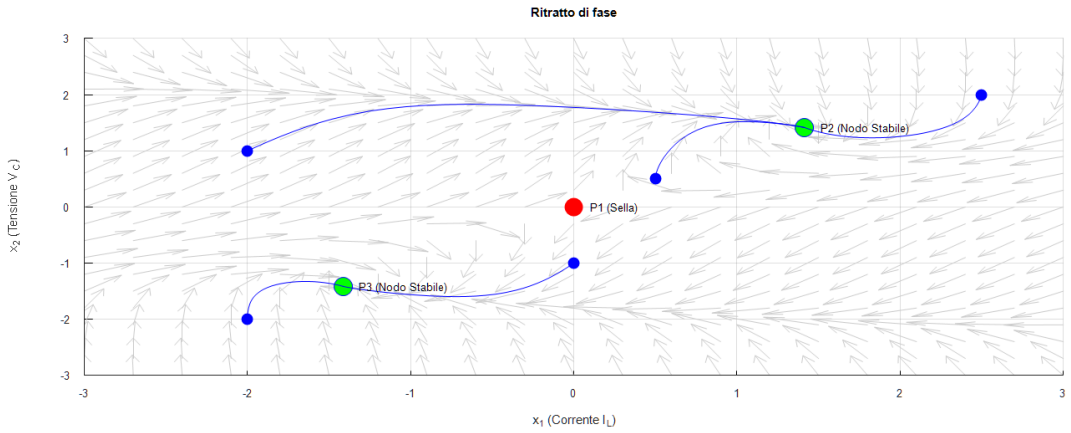

Figura 7: Ritratto di fase dei punti di equilibrio e traiettorie d'esempio

4.5 Codice

Lo script principale `tunnel_diode.m` del progetto definisce i parametri del circuito (R, L, C, α, β) e le equazioni differenziali non lineari del sistema. Lo script calcola analiticamente i tre punti di equilibrio, determina la matrice jacobiana simbolica e coordina l'analisi richiamando le altre due funzioni per lo studio della stabilità e la visualizzazione grafica.

```
clc;
clear all;
close all;

addpath('4')
pkg load control
pkg load symbolic

% Definisco le variabili simboliche per l'analisi
syms x1 x2

% Costanti del circuito
R = 1;
L = 1;
C = 1;
alpha = 3;
beta = 1;
```

```
% Definisco le equazioni di stato (il sistema non lineare f(x))
f1 = -R/L * x1 + 1/L * x2;
f2 = -1/C * x1 + alpha/C * x2 - beta/C * x2^3;
f = [f1; f2];

disp('Sistema non lineare dx/dt = f(x):');
disp(['dx1/dt = ', char(f1)]);
disp(['dx2/dt = ', char(f2)]);

% Trovo i punti (x1, x2) tali che f(x) = 0 (analiticamente)
P1 = [0; 0];
P2 = [sqrt(2); sqrt(2)];
P3 = [-sqrt(2); -sqrt(2)];

% Convertiamo in double per l'analisi numerica
P1_num = double(P1);
P2_num = double(P2);
P3_num = double(P3);

disp('\nTrovati 3 punti di equilibrio:');
disp('P1:');
disp(P1_num');
disp('P2:');
disp(P2_num');
disp('P3:');
disp(P3_num');

% Calcoliamo la matrice jacobiana
disp('')
disp('Calcolo matrice jacobiana');

J = jacobian(f, [x1, x2]);

disp('J(x1, x2) =');
```

```
disp(J);

% Analizzo e classifico i punti di equilibrio
disp('Analisi di stabilit  per ciascun equilibrio:');

analyze_equilibrium_point(J, x1, x2, P1_num)
analyze_equilibrium_point(J, x1, x2, P2_num)
analyze_equilibrium_point(J, x1, x2, P3_num)

% Creo una funzione "handle" per ode45
fun_diodo = @(t, x) [
    -R/L * x(1) + 1/L * x(2);
    -1/C * x(1) + alpha/C * x(2) - beta/C * x(2)^3
];
disp('Avvio simulazione e plot del ritratto di fase');

plot_vector_field(R, L, C, alpha, beta, P1_num, P2_num, P3_num,
   fun_diodo)
```

La funzione `analyze_equilibrium_point.m` prende in ingresso lo jacobiano simbolico e le coordinate di un punto di equilibrio specifico. Valuta numericamente la matrice nel punto, ne calcola gli autovalori e stampa la classificazione della stabilità basandosi sulla parte reale degli autovalori.

```
function analyze_equilibrium_point(J, x1, x2, P_num)
    J_P = double(subs(J, [x1, x2], P_num'));
    [V, D] = eig(J_P);
    poli_P = diag(D);

    disp('Equilibrio P = :' + P_num);
    disp('  Jacobiano J(P2):');
    disp(J_P);
    disp('  Autovalori (Poli):');
    disp(poli_P');

    % Classificazione
    if all(real(poli_P) < 0)
        disp('  -> STABILE (Nodo/Fuoco stabile)');
```

```
    elseif all(real(poli_P) > 0)
        disp('  -> INSTABILE (Nodo/Fuoco instabile)');
    elseif any(real(poli_P) > 0) && any(real(poli_P) < 0)
        disp('  -> INSTABILE (Punto di Sella)');
    else
        disp('  -> Caso critico');
    end
    disp(' ');
end
```

La funzione `plot_vector_field.m` è dedicata alla visualizzazione grafica. Genera il ritratto di fase del sistema: disegna il campo vettoriale, evidenzia i punti di equilibrio con colori diversi e simula numericamente alcune traiettorie rappresentative partendo da condizioni iniziali diverse per mostrare l'evoluzione temporale del sistema.

```
function plot_vector_field(R, L, C, alpha, beta, P1_num, P2_num,
   P3_num, fun_diodo)
    graphics_toolkit("gnuplot"); # fix per i grafici

    % Creiamo la griglia per il campo vettoriale (quiver plot)
    [X1_grid, X2_grid] = meshgrid(-3:0.3:3, -3:0.3:3);
    u = -R/L * X1_grid + 1/L * X2_grid;
    v = -1/C * X1_grid + alpha/C * X2_grid - beta/C * X2_grid.^3;

    % Normalizziamo le frecce per una migliore visualizzazione (
       evitiamo divisione per zero)
    denom = sqrt(u.^2 + v.^2) + eps;
    u_norm = u ./ denom;
    v_norm = v ./ denom;

    figure(1);
    hold on;

    % Plot del campo vettoriale
    quiver(X1_grid, X2_grid, u_norm, v_norm, 'Color', [0.8 0.8 0.8]);

    % Plot dei punti di equilibrio
```

```
plot(P1_num(1), P1_num(2), 'ro', 'MarkerSize', 10, '
    MarkerFaceColor', 'r');
text(P1_num(1)+0.1, P1_num(2), 'P1 (Sella)', 'Color', 'k');
plot(P2_num(1), P2_num(2), 'bo', 'MarkerSize', 10, '
    MarkerFaceColor', 'g');
text(P2_num(1)+0.1, P2_num(2), 'P2 (Nodo Stabile)', 'Color', 'k');
plot(P3_num(1), P3_num(2), 'bo', 'MarkerSize', 10, '
    MarkerFaceColor', 'g');
text(P3_num(1)+0.1, P3_num(2), 'P3 (Nodo Stabile)', 'Color', 'k');

% 3. Plot di alcune traiettorie
disp('Calcolo traiettorie di esempio');
t_span = [0 20];

% Condizioni iniziali di esempio
condizioni_iniziali = [
    0.5, 0.5;
    0.0, -1.0;
    2.5, 2.0;
    -2.0, 1.0;
    -2.0, -2.0
];

for i = 1:size(condizioni_iniziali, 1)
    [t, x_sim] = ode45(fun_diodo, t_span, condizioni_iniziali(i,:)
        );
    plot(x_sim(:, 1), x_sim(:, 2), 'b-', 'LineWidth', 1.5);

    % Plot del punto di partenza
    plot(x_sim(1,1), x_sim(1,2), 'bo', 'MarkerFaceColor', 'b');
end

% Impostazioni finali del grafico
title('Ritratto di fase');
xlabel('x_1 (Corrente I_L)');
```

```
    ylabel('x_2 (Tensione V_C)');
    axis([-3 3 -3 3]);
    grid on;
    hold off;
end
```

5 Massa-molla-smorzatore

Analisi sistemi massa-molla-smorzatore. Casi lineare e non-lineari seguenti:

$$M\ddot{x} = -k_1 x \pm k_2 x^3 - \beta_1 \dot{x} - \beta_2 \dot{x}^3$$

5.1 Introduzione

Il sistema massa-molla-smorzatore è un modello fisico fondamentale utilizzato per descrivere e analizzare il comportamento dinamico di sistemi meccanici soggetti a oscillazioni, vibrazioni e forze esterne. È alla base di molte applicazioni ingegneristiche, dall'analisi sismica delle strutture agli ammortizzatori dei veicoli.

5.2 Definizione del modello

Il sistema *massa-molla-smorzatore* può essere modellato come segue:

$$\begin{cases} \dot{x}_1 = x_2 \\ \dot{x}_2 = \frac{1}{M}(-k_1 x_1 \pm k_2 x_1^3 - \beta_1 x_2 - \beta_2 x_2^3) \end{cases} \tag{31}$$

dove x_1 rappresenta la posizione del carrello mentre $x_2 = \dot{x}_1$ la sua velocità. Definisco di tre casi:

- Approssimazione lineare.
- Versione non lineare con molla indurente ($-k_2 x_1^3$).
- Versione non lineare con molla ammorbidente ($+k_2 x_1^3$).

5.3 Analisi dei tre casi

5.3.1 Caso lineare

Il primo caso è quello in cui adotto un modello approssimato con dinamica lineare:

$$\begin{cases} \dot{x}_1 = x_2 \\ \dot{x}_2 = -\frac{k_1}{M} x_1 - \frac{\beta_1}{M} x_2 \end{cases} \tag{32}$$

L'unico punto x_e che annulla la derivata $\dot{x} = f(x_e) = 0$ è $(0, 0)$, l'origine. Lo jacobiano calcolato nel punto è il seguente:

$$J(x_e) = \begin{bmatrix} 0 & 0 \\ -1 & -0.5 \end{bmatrix} \tag{33}$$

Gli autovalori calcolati nel codice sono $\lambda_{1,2} = -0,25 \pm j0.968$, hanno parte reale strettamente negativa quindi l'equilibrio x_e è localmente asintoticamente stabile. Trattandosi di un sistema lineare

la stabilità dell'equilibrio è anche globale. Data la parte immaginaria non nulla, lo classifico come un fuoco stabile.

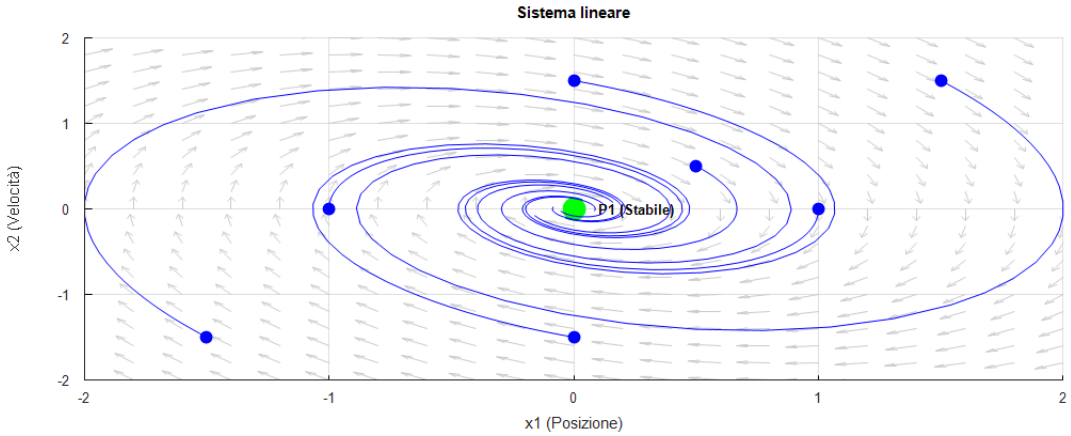

Figura 8: Traiettorie nel diagramma di fase nel caso lineare

5.3.2 Primo caso non lineare

Il secondo caso introduce una non-linearità sulla molla del tipo $-k_2x^3$ che rappresenta una molla indurente (la forza della molla aumenta all'aumentare dello spostamento). La forza è $F = -k_1x_1 - k_2x_1^3 - \beta_1x_2$.

$$\begin{cases} \dot{x}_1 = x_2 \\ \dot{x}_2 = \frac{1}{M}(-k_1x_1 - k_2x_1^3 - \beta_1x_2) \end{cases} \tag{34}$$

I punti di equilibrio si trovano ponendo $\dot{x}_1 = 0 \implies x_2 = 0$ e $\dot{x}_2 = 0 \implies -k_1x_1 - k_2x_1^3 = 0$. Questa equazione, $x_1(k_1 + k_2x_1^2) = 0$, assumendo $k_1, k_2 > 0$, ha come unica soluzione reale $x_1 = 0$. Anche in questo caso, l'unico punto di equilibrio è $x_e = (0, 0)$. La matrice jacobiana $J(x_e)$ è:

$$J(x_e) = \begin{bmatrix} 0 & 1 \\ \frac{-k_1-3k_2x_1^2}{M} & -\frac{\beta_1}{M} \end{bmatrix} \tag{35}$$

Calcolandola nell'equilibrio $x_e = (0, 0)$, otteniamo:

$$J(0,0) = \begin{bmatrix} 0 & 1 \\ -1 & -0.5 \end{bmatrix} \tag{36}$$

Questa è la stessa matrice del caso lineare. Gli autovalori sono $\lambda_{1,2} = -0.25 \pm j0.968$ di conseguenza, l'origine è un fuoco stabile. In questo caso, la non-linearità non ha alterato né il numero né la stabilità locale dei punti di equilibrio.

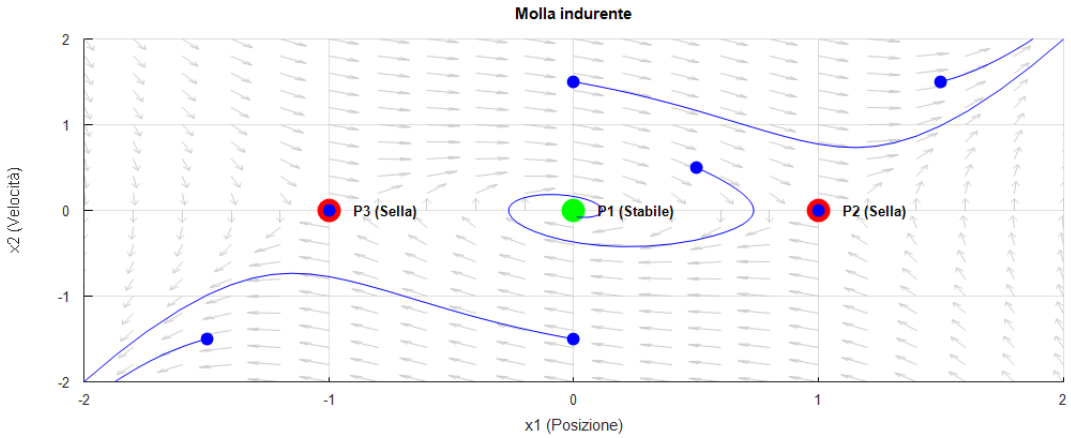

Figura 9: Traiettorie nel diagramma di fase nel caso con molla ammorbidente

5.3.3 Secondo caso non lineare

Il terzo caso introduce una non-linearità del tipo $+k_2x^3$ che rappresenta il caso di una molla ammorbidente (la forza della molla si riduce all'aumentare dello spostamento). La forza è $F = -k_1x_1 + k_2x_1^3 - \beta_1x_2$.

$$\begin{cases} \dot{x}_1 = x_2 \\ \dot{x}_2 = \frac{1}{M}(-k_1x_1 + k_2x_1^3 - \beta_1x_2) \end{cases} \tag{37}$$

I punti di equilibrio richiedono $x_2 = 0$ e $-k_1x_1 + k_2x_1^3 = 0$. Questa equazione, $x_1(k_2x_1^2 - k_1) = 0$, ha tre soluzioni reali:

$$x_1 = 0 \ \vee \ x_1 = \sqrt{k_1/k_2} = 1 \ \vee \ x_1 = -\sqrt{k_1/k_2} = -1 \tag{38}$$

Ci sono quindi tre punti di equilibrio: $P_1 = (0,0)$, $P_2 = (1,0)$, $P_3 = (-1,0)$. La matrice Jacobiana $J(x_e)$ per questo sistema è:

$$J(x_e) = \begin{bmatrix} 0 & 1 \\ \frac{-k_1+3k_2x_1^2}{M} & -\frac{\beta_1}{M} \end{bmatrix} \tag{39}$$

Devo analizzare lo jacobiano in ciascun punto:

$$J(0,0) = \begin{bmatrix} 0 & 1 \\ -1 & -0.5 \end{bmatrix} \tag{40}$$

Gli autovalori sono $\lambda_{1,2} = -0.25 \pm j0.968$. Come nei casi precedenti, l'origine è localmente asintoticamente stabile, un fuoco stabile. Poiché x_1 appare al quadrato nello Jacobiano, l'analisi è identica per P_2 e P_3.

$$J(\pm1,0) = \begin{bmatrix} 0 & 1 \\ \frac{-1+3(1)(\pm1)^2}{1} & -0.5 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 2 & -0.5 \end{bmatrix} \tag{41}$$

Gli autovalori sono $\lambda_1 \approx 1.19$ e $\lambda_2 \approx -1.69$. Avendo un autovalore con parte reale positiva e uno con parte reale negativa, entrambi P_2 e P_3 sono equilibri instabili, nello specifico punti di sella.

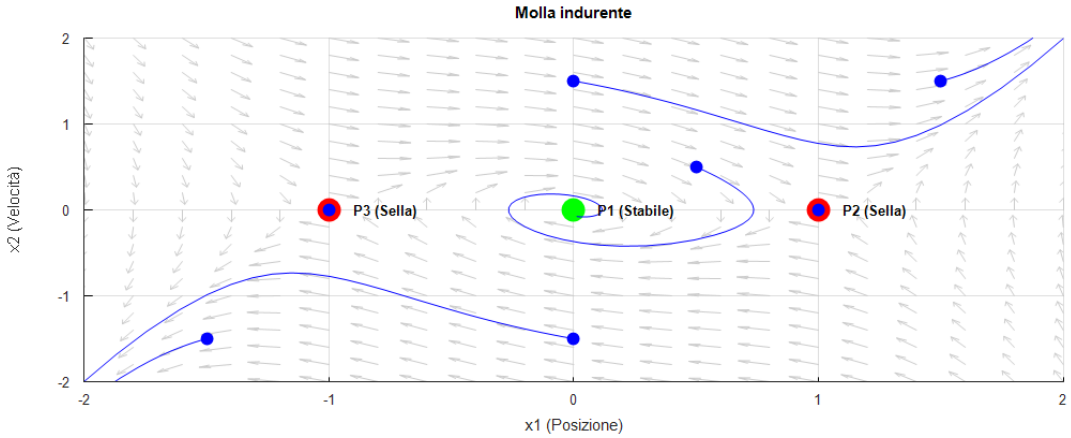

Figura 10: Traiettorie nel diagramma di fase nel caso con molla indurente

5.4 Codice

Lo script principale `mass_spring_damper.m` definisce i parametri fisici del sistema (massa M, rigidezza k_1, smorzamento β_1) e un parametro non lineare (k_2). Imposta le equazioni differenziali e le matrici jacobiane per tre casi diversi: lineare, molla "ammorbidente" ($k_2 > 0$) e molla "indurente" ($k_2 < 0$, anche se nel codice i segni sono invertiti rispetto alla nomenclatura standard, ma il concetto è quello di aggiungere una non linearità cubica). Infine, lancia l'analisi di stabilità e il plot del ritratto di fase per ciascun caso.

```
clc;
clear all;
close all;

pkg load control
pkg load symbolic

addpath('5')

% Parametri comuni
M = 1.0;
k1 = 1.0;
beta1 = 0.5; % Smorzamento leggero
```

```
% Parametro non lineare
k2 = 1.0;     % Termine della molla non lineare

% Caso 1: lineare
f_linear = @(t, x) [
    x(2);
    (1/M) * (-k1*x(1) - beta1*x(2))
];
J_linear = @(x) [0, 1; -k1/M, -beta1/M];

% Caso 2: non lineare (molla "ammorbidente")
f_hard = @(t, x) [
    x(2);
    (1/M) * (-k1*x(1) + k2*x(1)^3 - beta1*x(2))
];
J_hard = @(x) [0, 1; (-k1 + 3*k2*x(1)^2)/M, -beta1/M];

% Caso 3: non lineare (molla "indurente")
f_soft = @(t, x) [
    x(2);
    (1/M) * (-k1*x(1) - k2*x(1)^3 - beta1*x(2))
];
J_soft = @(x) [0, 1; (-k1 - 3*k2*x(1)^2)/M, -beta1/M];

% Condizioni iniziali
cond_iniziali = [
    1, 0;
    -1, 0;
    0, 1.5;
    0, -1.5;
    1.5, 1.5;
    -1.5, -1.5;
    0.5, 0.5
];
```

```
% Analisi caso 1
P_linear = [0, 0];
analyze_system(1, 'Sistema lineare', J_linear, P_linear);
plot_phase_portrait(1, f_linear, cond_iniziali);

% Analisi caso 2
P_soft = [0, 0];
analyze_system(2, 'Molla ammorbidente', J_soft, P_soft);
plot_phase_portrait(2, f_soft, cond_iniziali);

% Analisi caso 3
P_hard = [
    0, 0;
    sqrt(k1/k2), 0;
    -sqrt(k1/k2), 0
];
analyze_system(3, 'Molla indurente', J_hard, P_hard);
plot_phase_portrait(3, f_hard, cond_iniziali);
```

La funzione `analyze_system.m` esegue l'analisi di stabilità locale dei punti di equilibrio. Prende in ingresso la funzione che calcola lo jacobiano, i punti di equilibrio da analizzare e il numero della figura su cui disegnare. Per ogni punto, calcola gli autovalori dello Jacobiano linearizzato e determina se il punto è stabile (nodo/fuoco stabile) o instabile (punto di sella o altro), stampando i risultati a video e marcando i punti sul grafico con colori diversi (verde per stabile, rosso per instabile).

```
function analyze_system(fig_num, title_str, J_func, P_equilibri)
    figure(fig_num);
    clf; % Pulisce la figura
    title(title_str);
    hold on;
    disp(' ');
    disp(['Analisi: ', title_str]);

    for i = 1:size(P_equilibri, 1)
        P_num = P_equilibri(i, :);
        J_P = J_func(P_num);
```

```
        poli_P = eig(J_P);

        disp(['Equilibrio P', num2str(i), ' = (', num2str(P_num(1)), '
            , ', num2str(P_num(2)), ')']);
        disp('  Jacobiano J(P):');
        disp(J_P);
        disp('  Autovalori:');
        disp(poli_P');

        % Classificazione
        if all(real(poli_P) < 0)
            disp('  -> STABILE (Nodo/Fuoco stabile)');
            plot(P_num(1), P_num(2), 'go', 'MarkerSize', 10, '
                MarkerFaceColor', 'g');
            text(P_num(1)+0.1, P_num(2), ['P', num2str(i), ' (Stabile)
                '], 'Color', 'k', 'FontWeight', 'bold');
        elseif any(real(poli_P) > 0) && any(real(poli_P) < 0)
            disp('  -> INSTABILE (Punto di sella)');
            plot(P_num(1), P_num(2), 'ro', 'MarkerSize', 10, '
                MarkerFaceColor', 'r');
            text(P_num(1)+0.1, P_num(2), ['P', num2str(i), ' (Sella)'
                ], 'Color', 'k', 'FontWeight', 'bold');
        else
            disp('  -> INSTABILE (altri casi)');
            plot(P_num(1), P_num(2), 'rx', 'MarkerSize', 10, '
                MarkerFaceColor', 'r');
            text(P_num(1)+0.1, P_num(2), ['P', num2str(i), ' (
                Instabile)'], 'Color', 'k', 'FontWeight', 'bold');
        end
    end
end
```

La funzione `plot_phase_portrait.m` si occupa della visualizzazione grafica delle dinamiche del sistema (ritratto di fase). Genera una griglia di punti nello spazio delle fasi (posizione vs velocità) e calcola il vettore derivata in ogni punto per disegnare il campo vettoriale (le frecce grigie). Successivamente, simula numericamente l'evoluzione temporale del sistema partendo da diverse condizioni

iniziali predefinite usando `ode45` e sovrappone le traiettorie risultanti (linee blu) al campo vettoriale.

```
function plot_phase_portrait(fig_num, f_handle, condizioni_iniziali)
    graphics_toolkit("gnuplot"); # fix per i grafici
    figure(fig_num);

    % Creo la griglia per il campo vettoriale
    [X1_grid, X2_grid] = meshgrid(-2:0.2:2, -2:0.2:2);
    u = zeros(size(X1_grid));
    v = zeros(size(X2_grid));

    for i = 1:numel(X1_grid)
        dxdt = f_handle(0, [X1_grid(i); X2_grid(i)]);
        u(i) = dxdt(1);
        v(i) = dxdt(2);
    end

    % Normalizzo le frecce per una migliore visualizzazione
    norme = sqrt(u.^2 + v.^2);
    norme(norme == 0) = 1;
    quiver(X1_grid, X2_grid, u./norme, v./norme, 'AutoScaleFactor',
       0.5, 'Color', [0.8 0.8 0.8]);

    % Simulo e grafico le traiettorie
    t_span = [0 10];
    for i = 1:size(condizioni_iniziali, 1)
        [t, x_sim] = ode45(f_handle, t_span, condizioni_iniziali(i,:))
           ;
        plot(x_sim(:, 1), x_sim(:, 2), 'b-', 'LineWidth', 1.5);
        plot(x_sim(1,1), x_sim(1,2), 'bo', 'MarkerFaceColor', 'b', '
           MarkerSize', 5);
    end

    xlabel('x1 (Posizione)');
    ylabel('x2 (Velocit )');
    axis([-2 2 -2 2]);
```

```
    grid on;
    hold off;
end
```

6 Controllo adattativo

Controllo adattativo sistema scalare.

$$\dot{x} = ax + u$$

Stabilizzazione ed inseguimento di traiettoria. Analisi e simulazioni.

6.1 Introduzione

L'esercizio richiede di progettare un controllore adattativo che raggiunga due obiettivi:

- **Stabilizzazione**: portare lo stato $x(t)$ a convergere 0 a partire da una condizione arbitraria $x(0)$.
- **Inseguimento**: far si che lo stato $x(t)$ segua una traiettoria desiderata $x_d(t)$ minimizzando l'errore $e(t)$ di inseguimento nel tempo.

tenendo conto del parametro a sconosciuto.

6.2 Progetto del controllore con parametro noto

Se il parametro a fosse noto, potrei progettare direttamente il controllore, definendo la dinamica dell'errore di inseguimento:

$$\dot{e} = \dot{x} - \dot{x}_d = (ax + u) - \dot{x}_d \tag{42}$$

L' obiettivo è imporre una dinamica dell'errore asintoticamente stabile. Una scelta classica è ricondurlo ad una dinamica lineare del primo ordine:

$$\dot{e} = -Ke \quad (\text{con } K > 0) \tag{43}$$

In questo scenario ideale, l'errore decadrebbe esponenzialmente a zero con una costante di tempo $\tau = 1/K$. Se conoscessi il valore esatto di a, potrei calcolare un ingresso di controllo u^* che cancella la dinamica naturale del sistema e impone quella desiderata. Uguagliando la dinamica reale a quella desiderata:

$$ax + u^* - \dot{x}_d = -Ke \tag{44}$$

da cui ricavo, risolvendo per u^*:

$$u^* = \underbrace{-ax}_{\text{cancellazione}} + \underbrace{\dot{x}_d}_{\text{feedforward}} - \underbrace{Ke}_{\text{feedback stabilizzante}} \tag{45}$$

6.3 Legge di controllo adattativa

Nel caso in cui il parametro sia ignoto occorre adottare una sua stima tempo-variante $\hat{a}(t)$, che verrà aggiornata in tempo reale da una legge di adattamento. La legge di controllo diventa:

$$u(t) = -\hat{a}(t)x(t) + \dot{x}_d(t) - \lambda e(t) \tag{46}$$

Riscrivo anche la dinamica dell'errore con la nuova formulazione del controllo $u(t)$:

$$\dot{e} = ax + (-\hat{a}x + \dot{x}_d - \lambda e) - \dot{x}_d = \dot{e} = (a - \hat{a}(t))x - \lambda e = \dot{e}(t) = \tilde{a}(t)x(t) - \lambda e(t) \tag{47}$$

dove $\tilde{a}(t)$ è l'errore di stima di a. Il mio obiettivo è che il sistema che modella l'errore abbia uno stato che converge asintoticamente a 0, voglio, cioè, che sia un sistema asintoticamente stabile.

6.3.1 Funzione di Lyapunov

Per assicurarmi di ciò, posso sfruttare il **secondo criterio di Lyapunov** e costruire una funzione candidata $V(e, \tilde{a})$ che rappresenti l'energia dell'errore del sistema. Una scelta classica è la forma quadratica definita positiva che segue:

$$V(e, \tilde{a}) = e^2 + \frac{1}{\gamma}\tilde{a}^2 \tag{48}$$

dove $\gamma > 0$ è un parametro di progetto, detto *guadagno di adattamento*, che determina quanto velocemente reagisce l'algoritmo di stima agli errori. Calcolo la derivata di V rispetto al tempo ($\dot{V}$) per analizzare la stabilità:

$$\dot{V} = \frac{\partial V}{\partial e}\dot{e} + \frac{\partial V}{\partial \tilde{a}}\dot{\tilde{a}} = 2e\dot{e} + \frac{2}{\gamma}\tilde{a}\dot{\tilde{a}} = 2e(\tilde{a}x - \lambda e) - \frac{2}{\gamma}\tilde{a}\dot{\hat{a}} = -2\lambda e^2 + 2e\tilde{a}x - \frac{2}{\gamma}\tilde{a}\dot{\hat{a}} \tag{49}$$

Mi interessa raccogliere il termine da annullare che moltiplica $\tilde{a}$:

$$\dot{V} = -2\lambda e^2 + 2\tilde{a}\left(ex - \frac{1}{\gamma}\dot{\hat{a}}\right) \tag{50}$$

in modo da poter ridurre la derivata $\dot{V}$ al solo termine $-2\lambda e^2$ sempre negativo o nullo. Per farlo impongo:

$$ex - \frac{1}{\gamma}\dot{\hat{a}} = 0 \tag{51}$$

6.3.2 Legge di adattamento

Risolvendo per $\dot{\hat{a}}$, otteniamo la **legge di adattamento**:

$$\dot{\hat{a}}(t) = \gamma x(t)e(t) \tag{52}$$

A queste condizioni ottengo che $\dot{V} = -2\lambda e(t)^2 \leq 0 \ \forall t$ quindi:

- $V(t)$ è limitata quindi sia l'errore di inseguimento che $e(t)$ che quello di stima $\tilde{a}(t)$ lo sono.
- Applicando il Lemma di Barbalat, dato che $\dot{e}(t)$ è limitata (assumendo segnali limitati) posso affermare che $e(t) \to 0$ asintoticamente.

6.3.3 Convergenza del parametro

È importante notare che la stabilità asintotica dell'errore di inseguimento ($e \to 0$) non garantisce la convergenza del parametro stimato al valore vero ($\tilde{a} \to 0$). La convergenza della stima $\hat{a} \to a$ avviene solo se il segnale di riferimento soddisfa la condizione di **Eccitazione Persistente (PE)**:

- Nel caso di **stabilizzazione** ($x_d = 0$), il segnale decade a zero rapidamente. Quando $x \to 0$, la legge di aggiornamento $\dot{\hat{a}} = \gamma x e$ si annulla, e la stima si ferma a un valore costante errato. Questo non è un problema per l'obiettivo di controllo (l'errore e va comunque a zero), ma la stima non è accurata.
- Nel caso di **inseguimento** ($x_d = \sin(t)$), il segnale varia continuamente, soddisfacendo la condizione PE. Questo "eccita" costantemente l'algoritmo di adattamento, portando la stima $\hat{a}$ a convergere asintoticamente al valore vero a.

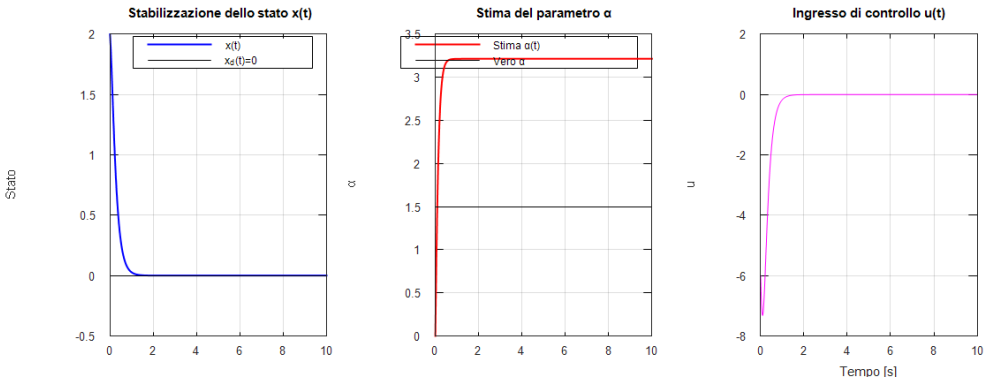

Figura 11: Stabilizzazione

Come si apprezza dalla Figura 11 la stima converge ad un valore inesatto come ci si aspettava.

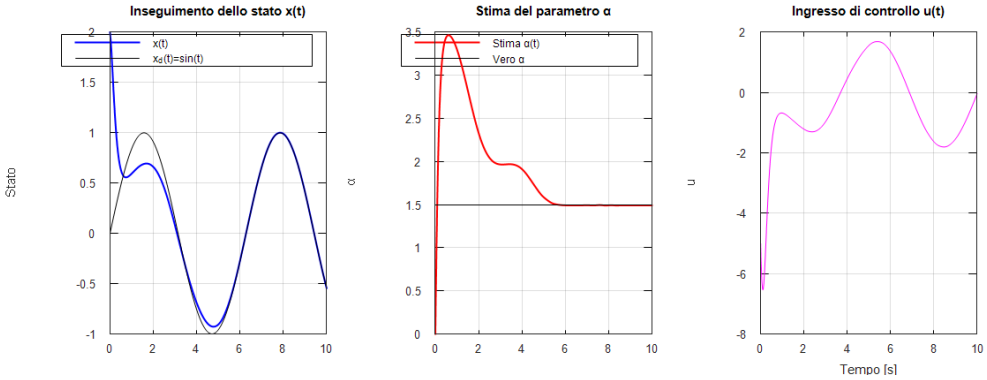

Figura 12: Inseguimento del riferimento

6.4 Codice

Lo script principale `adaptive_control.m` definisce i parametri del sistema ("vero" α e stima iniziale $\hat{\alpha}$), i parametri del controllore (λ e γ) e le condizioni di simulazione. Lo script esegue due simulazioni distinte usando ode45: una per la stabilizzazione a zero ($x_d = 0$) e una per l'inseguimento di una sinusoide ($x_d = \sin(t)$). Infine, ricostruisce i segnali di controllo e chiama la funzione di plotting.

```
clear all;
close all;
clc;

addpath('6')

% Parametri del sistema
global a_true lambda gamma

% Valore reale del parametro
a_true = 1.5;

% Parametri di progetto del controllore
lambda = 3.0;       % Velocit  di convergenza dell'errore
gamma  = 5.0;       % Guadagno di adattamento

% Parametri di simulazione
T_sim = 10;
dt = 0.01;
```

```
t_span = 0:dt:T_sim;

% Condizioni iniziali
x0 = 2;             % Stato iniziale del sistema
a_hat0 = 0;         % Stima iniziale del parametro 'a'
initial_state = [x0; a_hat0];

% Stabilizzazione
disp('Avvio simulazione di stabilizzazione');
type_ref = 1; % Stabilizzazione
[t1, y1] = ode45(@(t,y) system_dynamics(t, y, type_ref), t_span,
    initial_state);

% Inseguimento di traiettoria
disp('Avvio simulazione di inseguimento');
type_ref = 2; % Inseguimento sinusoide
[t2, y2] = ode45(@(t,y) system_dynamics(t, y, type_ref), t_span,
    initial_state);

% Ricostruzione segnali di controllo e riferimenti per i grafici
u1 = zeros(length(t1),1);
xd1 = zeros(length(t1),1);
for i=1:length(t1)
    [~, u_val, xd_val] = system_dynamics(t1(i), y1(i,:)', 1);
    u1(i) = u_val;
    xd1(i) = xd_val;
end

u2 = zeros(length(t2),1);
xd2 = zeros(length(t2),1);
for i=1:length(t2)
    [~, u_val, xd_val] = system_dynamics(t2(i), y2(i,:)', 2);
    u2(i) = u_val;
    xd2(i) = xd_val;
end
```

```
plot_res(y1, y2, xd1, xd2, t1, t2, u1, u2, a_true)
```

La funzione `system_dynamics.m` descrive la dinamica del sistema "aumentato" (stato del sistema di base e stato della stima del parametro). Implementa le equazioni differenziali:

- $\dot{x} = ax + u$ (dinamica del sistema scalare).
- $\dot{\hat{a}} = \gamma xe$ (legge di adattamento, regola di Lyapunov/MIT rule).Inoltre, calcola l'errore di inseguimento $e = x - x_d$ e applica la legge di controllo adattativa $u = -\hat{a}x + \dot{x}_d - \lambda e$.

```
function [dydt, u, x_d] = system_dynamics(t, y, type)
    global a_true lambda gamma

    x = y(1);       % Stato attuale
    a_hat = y(2);   % Stima attuale del parametro

    % Definizione traiettoria desiderata x_d(t) e sua derivata dx_d(t)
    if type == 1
        % Stabilizzazione
        x_d = 0;
        dx_d = 0;
    else
        % Inseguimento sinusoide
        x_d = sin(t);
        dx_d = cos(t);
    end

    % Calcolo Errore
    e = x - x_d;

    % Legge di controllo
    u = -a_hat * x + dx_d - lambda * e;

    % Dinamica del sistema
    dxdt = a_true * x + u;

    % Legge di adattamento
```

```
    da_hat_dt = gamma * x * e;
    dydt = [dxdt; da_hat_dt];
end
```

La funzione `plot_res.m` dedicata alla visualizzazione dei risultati. Genera due figure separate (una per la stabilizzazione, una per l'inseguimento), ciascuna contenente tre grafici:

- Confronto tra lo stato reale $x(t)$ e il riferimento desiderato $x_d(t)$.
- Evoluzione della stima del parametro $\hat{\alpha}(t)$ rispetto al valore vero.
- Andamento dell'ingresso di controllo $u(t)$.

```
function plot_res(y1, y2, xd1, xd2, t1, t2, u1, u2, a_true)
    graphics_toolkit("gnuplot"); % try setting gnuplot if available
    % Stabilizzazione
    figure(1);
    set(gcf, 'Name', 'Stabilizzazione', 'NumberTitle', 'off');
    subplot(1,3,1);
    plot(t1, y1(:,1), 'b', 'LineWidth', 2);
    hold on;
    plot(t1, xd1, 'k-', 'LineWidth', 1.5);
    grid on;
    title(' Stabilizzazione dello stato x(t)');
    legend('x(t)', 'x_d(t)=0');
    ylabel('Stato');

    subplot(1,3,2);
    plot(t1, y1(:,2), 'r', 'LineWidth', 2);
    hold on;
    plot([min(t1) max(t1)], [a_true a_true], 'k-', 'LineWidth', 1.5);
    grid on;
    title('Stima del parametro \alpha');
    legend('Stima \alpha(t)', 'Vero \alpha');
    ylabel('\alpha');

    subplot(1,3,3);
    plot(t1, u1, 'm', 'LineWidth', 1.5);
    grid on;
```

```
    title('Ingresso di controllo u(t)');
    xlabel('Tempo [s]');
    ylabel('u');

    % Inseguimento
    figure(2);
    set(gcf, 'Name', 'Inseguimento', 'NumberTitle', 'off');
    subplot(1,3,1);
    plot(t2, y2(:,1), 'b', 'LineWidth', 2);
    hold on;
    plot(t2, xd2, 'k-', 'LineWidth', 1.5);
    grid on;
    title('Inseguimento dello stato x(t)');
    legend('x(t)', 'x_d(t)=sin(t)');
    ylabel('Stato');

    subplot(1,3,2);
    plot(t2, y2(:,2), 'r', 'LineWidth', 2);
    hold on;
    plot([min(t2) max(t2)], [a_true a_true], 'k-', 'LineWidth', 1.5);
    grid on;
    title('Stima del parametro \alpha');
    legend('Stima \alpha(t)', 'Vero \alpha');
    ylabel('\alpha');

    subplot(1,3,3);
    plot(t2, u2, 'm', 'LineWidth', 1.5);
    grid on;
    title('Ingresso di controllo u(t)');
    xlabel('Tempo [s]');
    ylabel('u');
end
```

7 Controllo robot planare

Controllo robot planare a 2 link:

- *Regolazione spazio dei giunti e operativo.*
- *Tuning dei guadagni per assegnazione del tasso di convergenza locale.*
- *Inseguimento di traiettorie nello spazio dei giunti e nello spazio operativo con parametri noti.*
- *Inseguimento di traiettoria adattativo nello spazio dei giunti.*

7.1 Modello del robot planare a due link

Il robot a due bracci (RR) è un sistema non lineare e accoppiato. Le sue equazioni del moto si ottengono tramite il formalismo di Lagrange fornito durante le lezioni:

$$B(q)\ddot{q} + C(q,\dot{q})\dot{q} + F(\dot{q}) + g(q) = u \tag{53}$$

dove $q = [q_1, q_2]^T$ vettore delle coordinate dei giunti e le matrici sono definite come segue:

- $B(q)$: matrice di inerzia (simmetrica e definita positiva):

$$B(q) = \begin{bmatrix} \theta_1 + 2\theta_3\cos(q_2) & \theta_2 + \theta_3\cos(q_2) \\ \theta_2 + \theta_3\cos(q_2) & \theta_4 \end{bmatrix} \tag{54}$$

- $C(q,\dot{q})\dot{q}$: vettore delle forze di Coriolis e centrifughe:

$$C(q,\dot{q}) = \begin{bmatrix} -2\theta_3\dot{q}_2\sin(q_2) & -\theta_3\dot{q}_2\sin(q_2) \\ \theta_3\dot{q}_1\sin(q_2) & 0 \end{bmatrix} \tag{55}$$

- $F(\dot{q})$: vettore delle forze di attrito ai giunti:

$$F(\dot{q}) = \begin{bmatrix} F_1\dot{q}_1 \\ F_2\dot{q}_2 \end{bmatrix} \tag{56}$$

- $g(q)$: vettore delle forze gravitazionali.

$$g(q) = \begin{bmatrix} \theta_5\cos(q_1) + \theta_6\cos(q_1+q_2) \\ \theta_6\cos(q_1+q_2) \end{bmatrix} \tag{57}$$

- u: vettore delle coppie applicate ai giunti (ingresso di controllo).

I parametri reali forniti sono:

- $\theta_1 = I_1 + m_1\frac{l_1^2}{4} + m_2(\frac{l_2^2}{4} + l_1^2) + I_2 = 10.6125$

- $\theta_2 = I_2 + m_2 \frac{l_2^2}{4} = 0.85$
- $\theta_3 = m_2 l_1 \frac{l_2}{2} = 2.25$
- $\theta_4 = m_2 \frac{l_2^2}{2} + I_2 = 1.6$
- $\theta_5 = (m_1 \frac{l_1}{2} + m_2 l_1) g = 80.9325$
- $\theta_6 = m_2 \frac{l_2}{2} g = 14.7150$

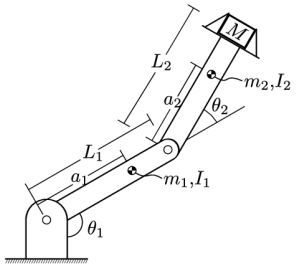

Figura 13: Schema di base di un robot RR planare

7.2 Regolazione spazio dei giunti e operativo

7.2.1 Spazio dei giunti

L'obiettivo è portare i giunti a una configurazione desiderata $q_d = [q_{d1}, q_{d2}]^T$ con velocità nulla. Facendo riferimento alla **teoria della stabilità di Lyapunov**, cerchiamo di progettare un controllore che renda il punto di equilibrio $(q = q_d,\ \dot{q} = 0)$ globalmente asintoticamente stabile. Una delle soluzioni più citate per problemi come questo è il controllore PD con compensazione della gravità che viene fornita nel testo dell'esercitazione. Scrivo quindi un controllo proporzionale e derivativo aggiungendo il termine $g(q)$:

$$u = K_P(q_d - q) - K_D\dot{q} + g(q) \tag{58}$$

Per dimostrare formalmente la stabilità, utilizzo il **secondo criterio di Lyapunov**. Definisco l'errore di posizione $\tilde{q} = q - q_d$. Considero la seguente funzione candidata di Lyapunov, che rappresenta l'energia totale del sistema a ciclo chiuso (energia cinetica ed energia potenziale):

$$V(\tilde{q}, \dot{q}) = \frac{1}{2}\dot{q}^T B(q)\dot{q} + \frac{1}{2}\tilde{q}^T K_P \tilde{q} \tag{59}$$

Con questa formulazione la funzione V risulta avere due caratteristiche fondamentali per la stabilità asintotica globale:

- **Definita positiva**: la matrice di inerzia $B(q)$ è sempre definita positiva per natura fisica. Il guadagno K_P è positivo per costruzione.

- **Radialmente illimitata**: poiché V è una forma quadratica, essa tende a infinito quando la norma dello stato $\tilde{q}$ tende a infinito.

Questa proprietà sono cruciale per assicurare la stabilità globale per il **teorema Barbashin-Krasovskii**.

Il passo successivo è calcolare la derivata di V lungo le traiettorie del sistema per verificarne la negativa definitezza:

$$\dot{V} = \dot{q}^T B(q)\ddot{q} + \frac{1}{2}\dot{q}^T \dot{B}(q)\dot{q} + \dot{q}^T K_P \dot{\tilde{q}} \tag{60}$$

Sostituisco ora l'accelerazione $\ddot{q}$ ricavandola dall'equazione dinamica del robot. Sapendo che $B(q)\ddot{q} = u - C(q,\dot{q})\dot{q} - F(\dot{q}) - g(q)$, sostituisco la legge di controllo u:

$$B(q)\ddot{q} = [K_P(-\tilde{q}) - K_D\dot{q} + g(q)] - C(q,\dot{q})\dot{q} - F(\dot{q}) - g(q) = -K_P\tilde{q} - K_D\dot{q} - C(q,\dot{q})\dot{q} - F(\dot{q}) \tag{61}$$

Ora inserisco questa espressione nella derivata $\dot{V}$:

$$\dot{V} = \dot{q}^T \left(-K_P\tilde{q} - K_D\dot{q} - C(q,\dot{q})\dot{q} - F(\dot{q})\right) + \frac{1}{2}\dot{q}^T \dot{B}(q)\dot{q} + \dot{q}^T K_P \dot{q} \tag{62}$$

Posso eliminare il termine $\dot{B}(q) - 2C(q,\dot{q})$ perché è *skew-symmetric*. Rimane quindi:

$$\dot{V} = -\dot{q}^T K_D \dot{q} - \dot{q}^T F(\dot{q}) \tag{63}$$

Poiché K_D è definita positiva e l'attrito $F(\dot{q})$ dissipa sempre energia (ha lo stesso segno della velocità), entrambi i termini sono non positivi:

$$\dot{V} \leq 0 \quad \forall(\tilde{q},\dot{q}) \tag{64}$$

La derivata è semidefinita negativa. Essa si annulla quando $\dot{q} = 0$, indipendentemente dal valore dell'errore di posizione $\tilde{q}$. Questo garantisce la stabilità semplice nel senso di Lyapunov (lo stato rimane limitato), ma non basta da solo a provare la convergenza asintotica a zero dell'errore di posizione.

Per dimostrare la stabilità asintotica, invochiamo il **principio di invarianza di LaSalle**, devo cercare il più grande insieme invariante contenuto nell'insieme S dove la derivata della funzione di Lyapunov è nulla:

$$S = \{(\tilde{q},\dot{q}) \in \mathbb{R}^{2n} : \dot{V}(\tilde{q},\dot{q}) = 0\} \tag{65}$$

Dall'espressione di $\dot{V}$, vedo che $\dot{V} = 0$ implica necessariamente che la velocità sia nulla:

$$\dot{V} = 0 \implies \dot{q} = 0 \tag{66}$$

Può il sistema rimanere indefinitamente in uno stato dove $\dot{q} = 0$ ma $\tilde{q} \neq 0$? Se il sistema rimane nell'insieme S, allora deve valere $\dot{q}(t) \equiv 0$ per tutto il tempo. Questo implica che anche l'accelerazione deve essere nulla: $\ddot{q}(t) \equiv 0$.

Sostituendo $\dot{q} = 0$ e $\ddot{q} = 0$ nell'equazione della dinamica a ciclo ottengo:

$$B(q) \cdot 0 + C(q, 0) \cdot 0 + F(0) + g(q) = K_P(q_d - q) - K_D \cdot 0 + g(q) \tag{67}$$

I termini inerziali, centrifughi e di attrito si annullano. I termini gravitazionali si elidono a vicenda. rimane solo l'equazione di equilibrio statico:

$$0 = K_P(q_d - q) = -K_P\tilde{q} \quad \text{e quindi} \quad \tilde{q} = 0 \tag{68}$$

Quindi, l'unico punto invariante in cui l'energia smette di decrescere è l'origine dello spazio di stato: $(\tilde{q} = 0, \dot{q} = 0)$. Pertanto, per il **teorema Barbashin-Krasovskii- LaSalle**, tutte le traiettorie convergeranno asintoticamente a questo equilibrio. Il sistema è Globalmente Asintoticamente Stabile.

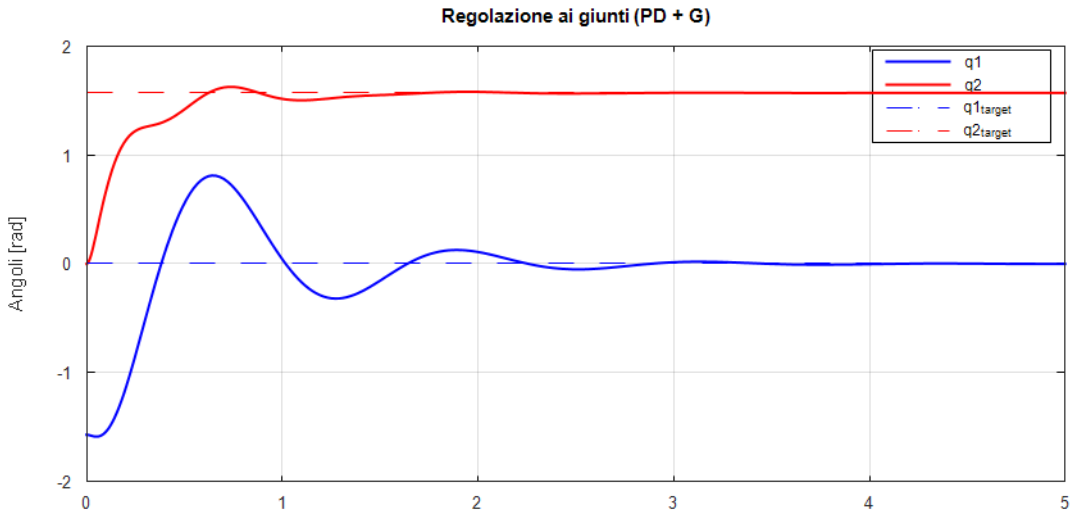

Figura 14: Controllo basato sullo spazio dei giunti

7.2.2 Spazio operativo

L'obiettivo è portare l'*end-effector* ad una posizione desiderata $x_d = [x_d, y_d]^T$. Definisco l'errore cartesiano: $\tilde{x} = x_d - x(q)$. La legge di controllo più usata è:

$$F_{task} = K_P\tilde{x} - K_D\dot{x} \tag{69}$$

Per applicare questa forza virtuale tramite i motori ai giunti, usiamo lo jacobiano:

$$u = J^T(q)F_{task} + g(q) = J^T(q)[K_P(x_d - x) - K_D J(q)\dot{q}] + g(q) \tag{70}$$

Posizione dell'end-effector (con lunghezze link l_1, l_2):

$$\begin{cases} x = l_1c_1 + l_2c_{12} \\ y = l_1s_1 + l_2s_{12} \end{cases} \tag{71}$$

Ricavo lo jacobiano analitico $J(q) = \frac{\partial x}{\partial q}$:

$$J(q) = \begin{bmatrix} -l_1 s_1 - l_2 s_{12} & -l_2 s_{12} \\ l_1 c_1 + l_2 c_{12} & l_2 c_{12} \end{bmatrix} \tag{72}$$

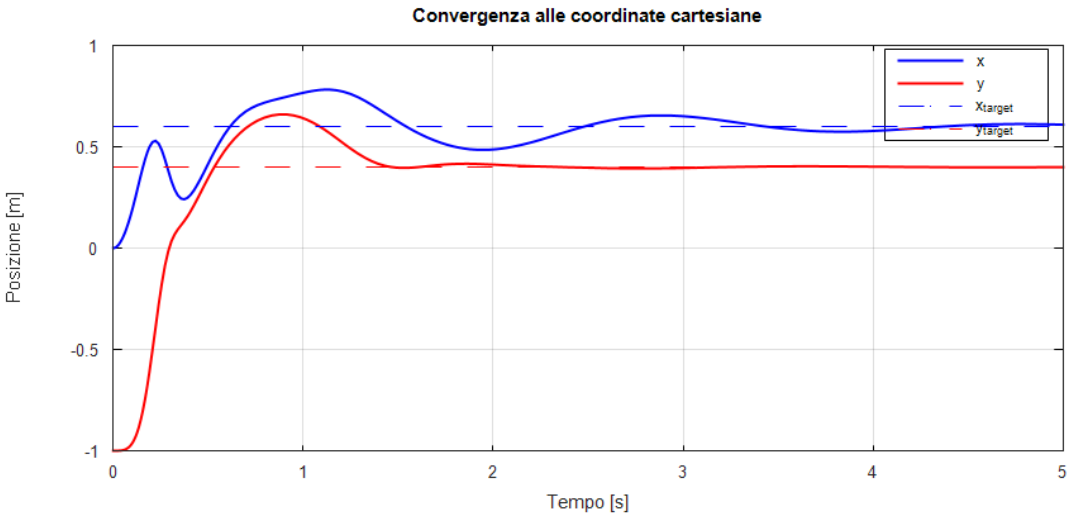

Figura 15: Controllo basato sullo spazio operativo

7.3 Tuning dei guadagni per assegnazione del tasso di convergenza locale

7.4 Inseguimento di traiettorie nello spazio dei giunti con parametri noti

Mentre la regolazione si preoccupa solo di portare q a q_{des} (dove $\dot{q}_{des} = \ddot{q}_{des} = 0$), l'inseguimento di traiettoria richiede che il robot assuma una specifica dinamica dello stato $q_d(t)$. Il solo controllo in *feedback* non è più sufficiente perché si rende necessario prevedere l'evoluzione dello stato, è necessaria una componente di *feedforward*.

La tecnica introdotta prende il nome di **Computed Torque** ed ha l'obiettivo di cancellare le nonlinearità del modello in maniera esatta. Il sistema robot, altamente non lineare e accoppiato, viene trasformato in un sistema lineare e disaccoppiato (una serie di doppi integratori $\ddot{e} = u_{PID}$). Questo modello prevede di scrivere l'errore come segue:

$$\ddot{e} + K_D \dot{e} + K_P e = 0 \tag{73}$$

In Figura 16 si vede come questo metodo garantisca una convergenza molto rapida sia in posizione che in velocità. Entrambi gli inseguimenti sembrano soddisfacenti già prima del secondo.

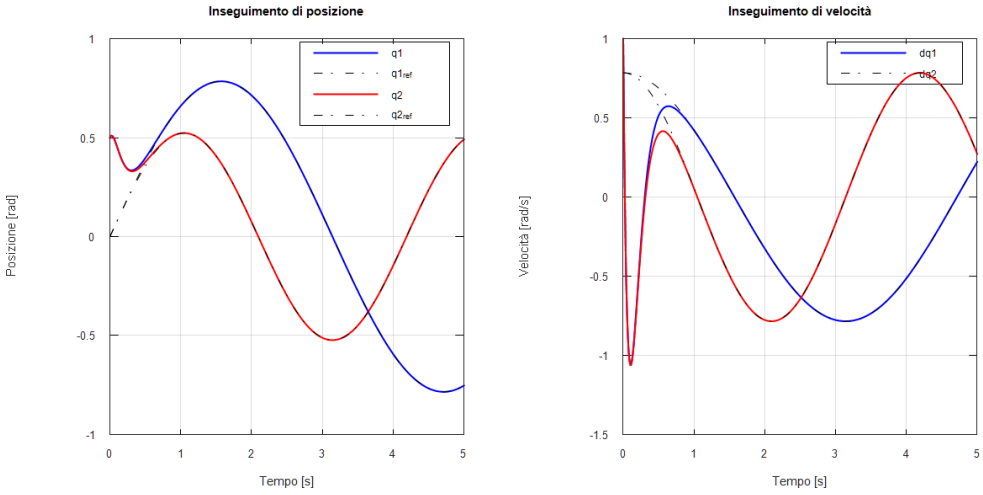

Figura 16: Inseguimento della traiettoria nello spazio dei giunti

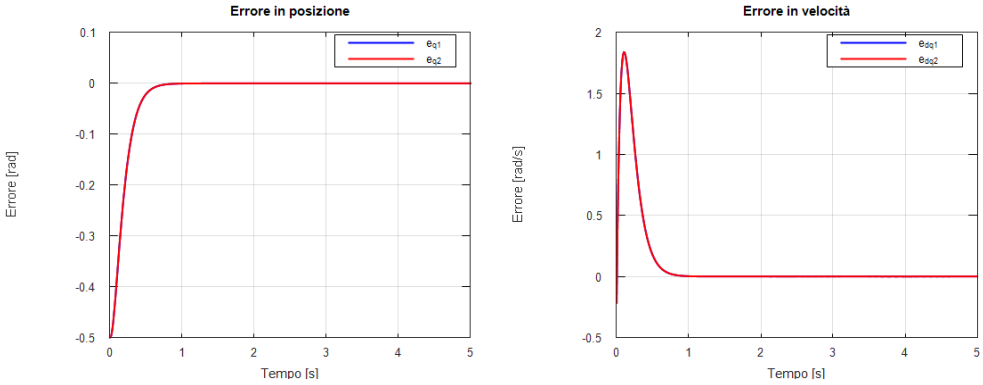

Figura 17: Errore di inseguimento della traiettoria nello spazio dei giunti

7.5 Inseguimento di traiettorie nello spazio operativo con parametri noti

A differenza del controllo ai giunti (dove il riferimento è θ_1, θ_2), qui il riferimento è una traiettoria cartesiana $x_d(t)$, nell'esempio ho usato un cerchio. I motori agiscono sui giunti (u), ma l'errore è definito nello spazio cartesiano ($e = x_d - x$). L'aggiunta rispetto alla sezione precedente dell'esercitazione è tradurre le traiettorie desiderate nello spazio cartesiano in controllo ai giunti.

7.5.1 Dinamica inversa nello spazio operativo

La strategia adottata è la linearizzazione esatta nello spazio operativo che risulta essere molto consigliata per approcciare questo tipo di problema. Partiamo dalla relazione cinematica differenziale dell'accelerazione:

$$\ddot{x} = J(q)\ddot{q} + \dot{J}(q, \dot{q})\dot{q} \tag{74}$$

Invertendo questa relazione (assumendo di non essere in singolarità), trovo l'accelerazione ai giunti necessaria per ottenere una certa accelerazione cartesiana $\ddot{x}$:

$$\ddot{q} = J^{-1}(\ddot{x} - \dot{J}\dot{q}) \tag{75}$$

Inserendo questo nel modello dinamico $B\ddot{q} + n = u$, ottengo la legge di controllo:

$$u = B(q)J^{-1}(q)(\underbrace{v_{cart}}_{\text{acc. cart. desiderata}} - \dot{J}\dot{q}) + C\dot{q} + F + g \tag{76}$$

7.5.2 Legge di controllo implementata

Il termine v_{cart} (accelerazione comandata) è calcolato come un PD in feedforward sull'errore cartesiano:

$$v_{cart} = \ddot{x}_d + K_D\dot{e} + K_P e \tag{77}$$

Questo impone all'errore cartesiano una dinamica lineare del secondo ordine:

$$\ddot{e}_x + K_D\dot{e}_x + K_P e_x = 0 \tag{78}$$

Un punto cruciale è l'implementazione di del termine $-\dot{J}\dot{q}$ che include lo jacobiano derivato che so rende necessario perché permette al controllore di compensare le accelerazioni centrifughe e di Coriolis che nascono dalla trasformazione cinematica stessa. L'inclusione di $\dot{J}$ rende l'inseguimento preciso anche a velocità elevate.

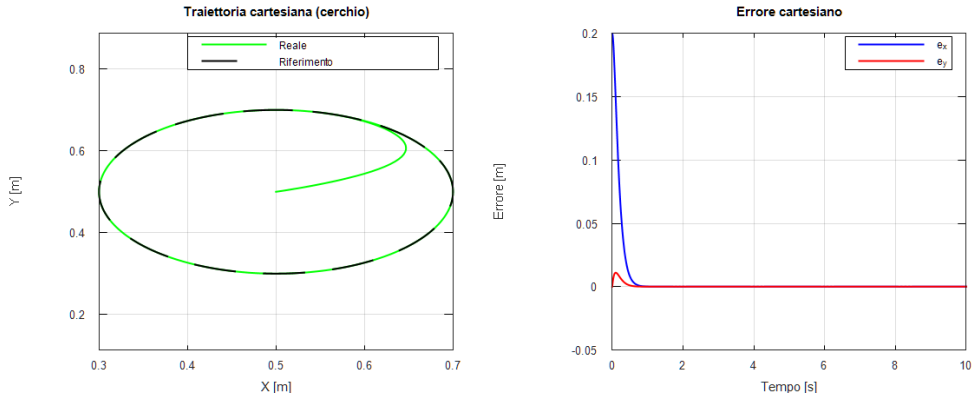

Figura 18: Inseguimento di traiettoria nello spazio operativo

7.6 Inseguimento di traiettoria adattativo nello spazio dei giunti

7.6.1 Controllo adattativo di Slotine-Li

Nel *Computed Torque Control* standard che ho usato per l'inseguimento di traiettoria, la legge di controllo si basa sulla conoscenza esatta delle matrici B, C, F e g ma se queste sono ignote non

è possibile cancellare esattamente la componente non lineare. La soluzione a questo problema è il **controllo adattativo di Slotine-Li** che suggerisce di riscrivere la dinamica del robot come segue:

$$B(q)\ddot{q} + C(q,\dot{q})\dot{q} + F\dot{q} + g(q) = Y(q,\dot{q},\ddot{q}) \cdot \theta \tag{79}$$

dove θ è un vettore costante contenente i parametri fisici (masse, inerzie, ecc.) e Y è una matrice chiamata **regressore**, che dipende solo da $q, \dot{q}, \ddot{q}$ che sono quantità note o misurabili. Invece di usare i parametri veri θ (ignoti), uso una stima $\hat{\theta}(t)$ che cambia nel tempo:

$$u = Y(q,\dot{q},\dot{q}_r,\ddot{q}_r)\hat{\theta} - K_D s \tag{80}$$

dove $\dot{q}_r = \dot{q}_d - \Lambda\tilde{q}$ è la velocità di riferimento virtuale, $s = \dot{q} - \dot{q}_r = \dot{\tilde{q}} + \Lambda\tilde{q}$ è la superficie di scorrimento, una misura composita dell'errore .Il primo termine $Y\hat{\theta}$ cerca di compensare la dinamica usando la stima attuale. Il secondo termine $(-K_D s)$ è un feedback robusto che spinge l'errore a zero.

7.6.2 Legge di adattamento

Per aggiornare la stima $\hat{\theta}$ uso una legge basata sul gradiente dell'errore:

$$\dot{\hat{\theta}} = -\Gamma Y^T s \tag{81}$$

Dove Γ è una matrice diagonale dei tassi di apprendimento (learning rate).

7.6.3 Implementazione

Per effettuare la stima effettuiamo una regressione che ha l'obiettivo di ricostruire completamente la matrice Y che sostituisce idealmente i parametri che abbiamo ammesso di non conoscere. Anche la dinamica del robot deve essere riscritta per calcolare l'errore composito che il regressore deve annullare.

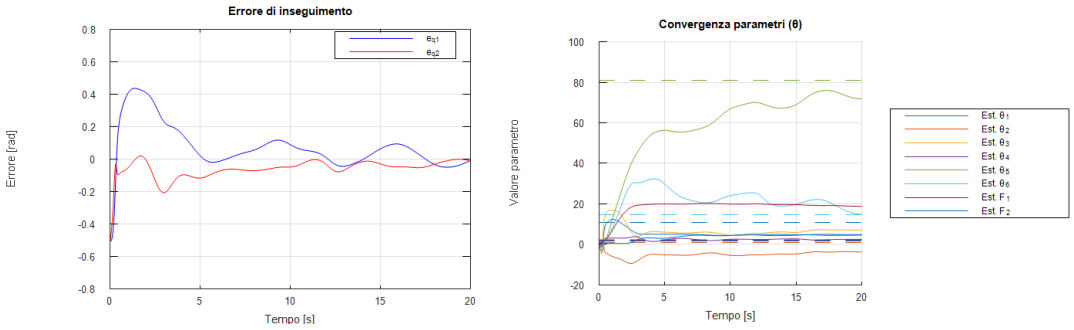

Figura 19: Errore di inseguimento (sinistra) e andamento della stima dei parametri (destra)

7.7 Codice

Lo script principali `rr_robot_control.m` ha diversi compiti: inizializzazione e definizione del modello, regolazione nello spazio dei giunti, regolazione nello spazio operativo, inseguimento di traiettoria nello spazio dei giunti, inseguimento di traiettoria nello spazio operativo e controllo adattativo.

```
clear all;
close all;
clc;

addpath('7')

global theta geom

% Parametri dinamici
theta = [10.6125; 0.85; 2.25; 1.6; 80.9325; 14.7150];

% Parametri geometrici (Assunti per la cinematica)
geom.l1 = 0.5; % Lunghezza link 1 [m]
geom.l2 = 0.5; % Lunghezza link 2 [m]

%% CONTROLLO NELLO SPAZIO DEI GIUNTI
disp('\nRegolazione spazio giunti');

% Target: Vogliamo portare il robot in una configurazione specifica
q_des = [0; pi/2];

% Condizione iniziale (fermo in posizione diversa: braccio gi )
q0_joint = [-pi/2; 0; 0; 0];

% Simulazione
T_sim = 5;

% Chiamata a ode45 per risolvere la dinamica controllata ai giunti
[t_j, y_j] = ode45(@(t,y) robot_dynamics_joint(t, y, q_des), [0 T_sim
    ], q0_joint);
```

```
plot_res_joint(t_j, y_j, q_des, T_sim)

%% CONTROLLO NELLO SPAZIO OPERATIVO
disp('\nRegolazione spazio operativo');

% Target in coordinate cartesiane (x, y)
x_des = 0.6;
y_des = 0.4;
target_pos = [x_des; y_des];

% Condizione iniziale
q0_op = [-pi/2; 0; 0; 0];

% Chiamata a ode45 per risolvere la dinamica controllata nello spazio
   operativo
[t_op, y_op] = ode45(@(t,y) robot_dynamics_operational(t, y,
   target_pos), [0 T_sim], q0_op);

% Visualizzazione dei risultati
plot_res_op(y_op, target_pos, t_op);

%% CONTROLLO IN TRAIETTORIA NELLO SPAZIO DEI GIUNTI
disp('\nInseguimento di traiettorie nello spazio dei giunti con
   parametri noti');

% Definizione delle traiettorie desiderata
A_traj = [pi/4; pi/6];      % Ampiezze [rad]
w_traj = [1.0; 1.5];        % Frequenze [rad/s]

% Condizione iniziale
q0 = [0.5; 0.5];
dq0 = [1; 1];
x0 = [q0; dq0];

% Simulazione
```

```
T_sim = 5;
disp('Avvio simulazione Computed Torque...');
[t, y] = ode45(@(t,y) robot_dynamics_computed_torque(t, y, A_traj,
    w_traj), [0 T_sim], x0);

% Risultati
q = y(:, 1:2);
dq = y(:, 3:4);

% Ricostruzione riferimenti per il plot
q_des = zeros(length(t), 2);
dq_des = zeros(length(t), 2);
for i=1:length(t)
    q_des(i,:) = (A_traj .* sin(w_traj*t(i)))';
    dq_des(i,:) = (A_traj .* w_traj .* cos(w_traj*t(i)))';
end

% Plot
plot_res_traj(t, q, dq, q_des, dq_des)

%% CONTROLLO IN TRAIETTORIA NELLO SPAZIO OPERATIVO
disp('Inseguimento traiettoria nello spazio operativo');

% Definizione Cerchio
traj_data.Center = [0.5; 0.5];
traj_data.Radius = 0.2;
traj_data.Omega  = 1.5;

% Partenza
q0_op_tr = [0; pi/2];
dq0_op_tr = [0; 0];
x0_op_tr = [q0_op_tr; dq0_op_tr];

% Simulazione
T_sim_op = 10;
```

```
[t_opt, y_opt] = ode45(@(t,y) robot_dynamics_op_track(t, y, traj_data)
    , [0 T_sim_op], x0_op_tr);

% Analisi risultati e plot
x_real = zeros(length(t_opt), 2);
x_des_tr = zeros(length(t_opt), 2);
e_x = zeros(length(t_opt), 2);

for i=1:length(t_opt)
    % Posizione reale (Cinematica Diretta)
    [pos_real, ~] = get_kinematics(y_opt(i,1:2)');
    x_real(i,:) = pos_real';

    % Ricalcolo riferimento al tempo t_opt(i)
    C = traj_data.Center;
    R = traj_data.Radius;
    w = traj_data.Omega;
    t_val = t_opt(i);
    xd = C + [R*cos(w*t_val); R*sin(w*t_val)];
    x_des_tr(i,:) = xd';
    e_x(i,:) = (xd - pos_real)';
end

%  Plot dei risultati
plot_res_op_traj(t_opt, x_real, x_des_tr, e_x)

%% CONTROLLO ADATTIVO
disp('Inseguimento adattativo (parametri ignoti)...');

% Parametri di attrito "reali"
F_real = [2; 2];
T_adapt = [0 20]

% Parametri Adattativi
Lambda = diag([5, 5]);      % Guadagno sulla posizione
```

```
Kd_adapt = diag([20, 20]);  % Guadagno sulla "sliding variable"
Gamma = eye(8) * 2;         % Velocit  di apprendimento parametri
Gamma(5,5) = 10;
Gamma(6,6) = 10;

% Condizioni iniziali: stato + stima parametri
theta_hat0 = zeros(8, 1);
y0_adapt = [x0; theta_hat0]; % [q; dq; theta_hat]
[t_ad, y_ad] = ode45(@(t,y) robot_dynamics_adaptive(t, y, A_traj,
    w_traj, Lambda, Gamma, Kd_adapt), T_adapt, y0_adapt);

% Estrazione dati
q_ad = y_ad(:, 1:2);
dq_ad = y_ad(:, 3:4);
theta_hist = y_ad(:, 5:end);

% Ricostruzione riferimento
q_des_ad = zeros(length(t_ad), 2);
for i=1:length(t_ad)
    q_des_ad(i,:) = (A_traj .* sin(w_traj*t_ad(i)))';
end

% Vettore parametri veri completo
theta_real_full = [theta; F_real];
plot_res_adaptive(t_ad, q_ad, dq_ad, theta_hist, q_des_ad,
    theta_real_full);
```

La funzione `robot_dynamics_joint.m` implementa un Controllo PD con compensazione della gravità nello spazio dei giunti (Joint Space Control).

```
function dy = robot_dynamics_joint(t, y, q_des)
    % Assicuriamoci che y sia un vettore colonna
    y = y(:);
    q = y(1:2);
    dq = y(3:4);

    % Recuperiamo le matrici dinamiche
```

```
    [B, C, g, F] = get_dynamic_matrices(q, dq);

    % Impostazione dei guadagni
    Kp = diag([300, 300]);  % Guadagno proporzionale
    Kd = diag([30, 30]);    % Guadagno derivativo

    e_q = q_des - q;        % Errore di posizione ai giunti

    % Controllo PD + Compensazione Gravit  (g)
    u = Kp * e_q - Kd * dq + g;

    % Dinamica
    ddq = B \ (u - C*dq - F - g);
    dy = [dq; ddq];
end
```

La funzione `robot_dynamics_operational.m` implementa un Controllo PD con compensazione della gravità nello spazio operativo (Operational Space Control).

```
function dy = robot_dynamics_operational(t, y, x_des)
    % Assicuriamoci che y sia un vettore colonna (4x1)
    y = y(:);
    q = y(1:2); dq = y(3:4);

    % Calcolo matrici dinamiche (B, C, g, F)
    [B, C, g, F] = get_dynamic_matrices(q, dq);

    % Calcolo cinematica e jacobiano
    [x_curr, J] = get_kinematics(q);

    % Legge di controllo nello spazio operativo
    e_x = x_des - x_curr;

    % Velocit  cartesiana reale
    dx = J * dq;

    % Guadagni del controllore
```

```
    Kp = diag([500, 500]);
    Kd = diag([50, 50]);

    % Forza virtuale cartesiana richiesta (2x1)
    F_task = Kp * e_x - Kd * dx;

    % Mappatura in coppie ai giunti tramite J trasposto +
       Compensazione Gravit
    u = J' * F_task + g;

    % 4. Dinamica diretta del robot
    ddq = B \ (u - C*dq - F - g);
    dy = [dq; ddq];
end
```

La funzione `robot_dynamics_computed_torque.m` implementa il controllo Computed Torque per l'inseguimento di traiettoria nello spazio dei giunti.

```
function dy = robot_dynamics_computed_torque(t, y, A_traj, w_traj)
    y = y(:);
    q = y(1:2);
    dq = y(3:4);

    % Generazione traiettoria di riferimento
    qd = A_traj .* sin(w_traj*t);
    dqd = A_traj .* w_traj .* cos(w_traj*t);
    ddqd = -A_traj .* w_traj.^2 .* sin(w_traj*t);

    % Calcolo matrici dinamiche
    [B, C, g, F] = get_dynamic_matrices(q, dq);

    % Legge di controllo Computed Torque
    e = qd - q;
    de = dqd - dq;

    % Tuning PD
    wn = 10; % Banda passante [rad/s]
```

```
    Kp = diag([wn^2, wn^2]);
    Kd = diag([2*wn, 2*wn]);

    % Ingresso ausiliario (doppio integratore stabilizzato)
    u_aux = ddqd + Kd*de + Kp*e;

    % Linearizzazione tramite feedback (Calcolo Coppia)
    u = B * u_aux + C*dq + F + g;

    % Dinamica diretta
    ddq = B \ (u - C*dq - F - g);
    dy = [dq; ddq];
end
```

La funzione `robot_dynamics_op_track.m` implementa un controllo a Coppia Calcolata nello spazio operativo.

```
function dy = robot_dynamics_op_track(t, y, traj_data)
    y = y(:);
    q = y(1:2);
    dq = y(3:4);

    % Riferimenti cartesiani (cerchio)
    C = traj_data.Center;
    R = traj_data.Radius;
    w = traj_data.Omega;

    xd = C + [R*cos(w*t); R*sin(w*t)];
    dxd = [-R*w*sin(w*t); R*w*cos(w*t)];
    ddxd = [-R*w^2*cos(w*t); -R*w^2*sin(w*t)];

    % Cinematica attuale
    [x, J] = get_kinematics(q);

    % Calcolo della dJ
    dJ = get_jacobian_dot(q, dq);
    dJ_dq = dJ * dq;
```

```
    dx = J * dq;

    % Legge di controllo operativa
    e = xd - x;
    de = dxd - dx;

    % Tuning PD cartesiano
    wn = 10;
    Kp = diag([wn^2, wn^2]);
    Kd = diag([2*wn, 2*wn]);

    % Accelerazione cartesiana virtuale
    v_cart = ddxd + Kd*de + Kp*e;

    % Mappatura nello spazio dei giunti
    ddq_ref = J \ (v_cart - dJ_dq);

    % Dinamica inversa (Computed Torque)
    [B, C_mat, g, F] = get_dynamic_matrices(q, dq);

    tau = B * ddq_ref + C_mat*dq + F + g;

    % Dinamica diretta
    ddq = B \ (tau - C_mat*dq - F - g);
    dy = [dq; ddq];
end
```

La funzione `robot_dynamics_adaptive.m` implementa il Controllo Adattativo di Slotine-Li.

```
function dy = robot_dynamics_adaptive(t, y_aug, A_traj, w_traj, Lambda
    , Gamma, Kd)
    % Stato aumentato
    q = y_aug(1:2);
    dq = y_aug(3:4);
    theta_hat = y_aug(5:end);
```

```
    % Generazione traiettoria
    qd = A_traj .* sin(w_traj*t);
    dqd = A_traj .* w_traj .* cos(w_traj*t);
    ddqd = -A_traj .* w_traj.^2 .* sin(w_traj*t);

    % Variabili "sliding" di Slotine-Li ---
    e = q - qd;

    % Velocit  e accelerazione di riferimento "virtuali"
    dqr = dqd - Lambda * e;
    ddqr = ddqd - Lambda * (dq - dqd);

    % Superficie di scorrimento s = dq - dqr = de + Lambda*e
    s = dq - dqr;

    % Calcolo Regressore Y(q, dq, dqr, ddqr)
    Y = get_regressor(q, dq, dqr, ddqr);

    % Legge di controllo: Feedforward Adattativo (Y*theta_hat) +
       Feedback PD (Kd*s)
    u = Y * theta_hat - Kd * s;

    % Legge di adattamento (Update Parametri)
    d_theta_hat = -Gamma * Y' * s;

    % Utilizziamo i parametri "veri" (ignoti al controllore) per la
       simulazione
    [B, C, g, F] = get_dynamic_matrices(q, dq);

    % Dinamica diretta: B*ddq + C*dq + F + g = tau
    ddq = B \ (u - C*dq - F - g);

    % Derivata dello stato aumentato
    dy = [dq; ddq; d_theta_hat];
end
```

La funzione `get_dynamic_matrices.m` restituisce le matrici che compongono il modello dinamico del robot nella forma: $B(q)\ddot{q} + C(q,\dot{q})\dot{q} + g(q) + F\dot{q} = \tau$.

```
function [B, C, g, F] = get_dynamic_matrices(q, dq)
    global theta
    t1=theta(1);
    t2=theta(2);
    t3=theta(3);
    t4=theta(4);
    t5=theta(5);
    t6=theta(6);

    c1 = cos(q(1));
    c2 = cos(q(2));
    s2 = sin(q(2));
    c12 = cos(q(1)+q(2));

    B = [t1 + 2*t3*c2, t2 + t3*c2;
         t2 + t3*c2,   t4];

    C = [-2*t3*dq(2)*s2, -t3*dq(2)*s2;
               t3*dq(1)*s2,    0];

    g = [t5*c1 + t6*c12;
         t6*c12];

    F = [2*dq(1); 2*dq(2)];
end
```

La funzione `get_kinematics.m` calcola la cinematica diretta e lo Jacobiano analitico del robot.

```
function [x, J] = get_kinematics(q)
    global geom
    l1 = geom.l1;
    l2 = geom.l2;
    q1 = q(1);
    q2 = q(2);
```

```
    % Funzioni trigonometriche ausiliarie
    s1 = sin(q1); c1 = cos(q1);
    s12 = sin(q1+q2); c12 = cos(q1+q2);

    % Cinematica diretta
    x_pos = l1*c1 + l2*c12;
    y_pos = l1*s1 + l2*s12;
    x = [x_pos; y_pos];

    % Jacobiano Analitico
    J = [-l1*s1 - l2*s12,   -l2*s12;
          l1*c1 + l2*c12,    l2*c12];
end
```

La funzione `get_jacobian_dot.m` calcola la derivata temporale della matrice Jacobiana ($\dot{J}$).

```
function dJ = get_jacobian_dot(q, dq)
    global geom
    l1 = geom.l1; l2 = geom.l2;

    q1 = q(1); q2 = q(2);
    dq1 = dq(1); dq2 = dq(2);
    s1 = sin(q1); c1 = cos(q1);
    s12 = sin(q1+q2); c12 = cos(q1+q2);

    % Termine (1,1): d(-l1*s1 - l2*s12)/dt
    dJ11 = -l1*c1*dq1 - l2*c12*(dq1+dq2);

    % Termine (1,2): d(-l2*s12)/dt
    dJ12 = -l2*c12*(dq1+dq2);

    % Termine (2,1): d(l1*c1 + l2*c12)/dt
    dJ21 = -l1*s1*dq1 - l2*s12*(dq1+dq2);

    % Termine (2,2): d(l2*c12)/dt
    dJ22 = -l2*s12*(dq1+dq2);
```

```
    % Assemblo lo jacobiano
    dJ = [dJ11, dJ12;
          dJ21, dJ22];
end
```

La funzione `get_regressor.m` calcola la matrice di regressione $Y(q, \dot{q}, \dot{q}_r, \ddot{q}_r)$.

```
function Y = get_regressor(q, dq, dqr, ddqr)
    % Costruisce la matrice Y tale che Y*Theta = B*ddqr + C*dqr + g +
       F

    q1 = q(1); q2 = q(2);
    dq1 = dq(1); dq2 = dq(2);
    dqr1 = dqr(1); dqr2 = dqr(2);
    ddqr1 = ddqr(1); ddqr2 = ddqr(2);

    c1 = cos(q1); c2 = cos(q2); s2 = sin(q2); c12 = cos(q1+q2);

    % t1: coeff di ddqr1
    Y11 = ddqr1;
    % t2: coeff di ddqr2
    Y12 = ddqr2;
    % t3: Termini di inerzia accoppiata e Coriolis
    Y13 = 2*c2*ddqr1 + c2*ddqr2 - 2*s2*dq2*dqr1 - s2*dq2*dqr2;
    % t4: 0
    Y14 = 0;
    % t5: c1 (Gravit  link 1)
    Y15 = c1;
    % t6: c12 (Gravit  link 2)
    Y16 = c12;
    % f1: dqr1 (Attrito viscoso giunto 1)
    Y17 = dqr1;
    % f2: 0
    Y18 = 0;

    Y21 = 0;
    Y22 = ddqr1;
```

```
    Y23 = c2*ddqr1 + s2*dq1*dqr1;
    Y24 = ddqr2;
    Y25 = 0;
    Y26 = c12;
    Y27 = 0;
    Y28 = dqr2;

    Y = [Y11, Y12, Y13, Y14, Y15, Y16, Y17, Y18;
         Y21, Y22, Y23, Y24, Y25, Y26, Y27, Y28];
end
```

Le funzioni `plot_res_joint.m`, `plot_res_joint.m`, `plot_res_op.m`, `plot_res_traj.m`, `plot_res_op_traj.m` e `plot_res_adaptive.m` sono le funzioni che mostrano l'inseguimento delle posizioni nello spazio dei giunti e in quello operativo e l'inseguimento di traiettoria nello spazio dei giunti e in quello operativo.

```
function plot_res_joint(t_j, y_j, q_des, T_sim)
    graphics_toolkit("gnuplot"); # fix per i grafici
    figure('Name', 'Controllo nello spazio dei giunti');
    plot(t_j, y_j(:,1), 'b', 'LineWidth', 2);
    hold on;
    plot(t_j, y_j(:,2), 'r', 'LineWidth', 2);
    plot([0 T_sim], [q_des(1) q_des(1)], 'b--');
    plot([0 T_sim], [q_des(2) q_des(2)], 'r--');
    grid on;
    ylabel('Angoli [rad]');
    legend('q1', 'q2', 'q1_{target}', 'q2_{target}');
    title('Regolazione ai giunti (PD + G)');
end

function plot_res_op(y, target_pos, t);
    graphics_toolkit("gnuplot"); # fix per i grafici
    % Estrazione coordinate target
    x_des = target_pos(1);
    y_des = target_pos(2);
    T_sim = t(end);

    q1 = y(:,1);
    q2 = y(:,2);
```

```
    x_traj = zeros(length(t), 2);

    % Ricostruzione traiettoria
    for i=1:length(t)
        [pos, ~] = get_kinematics(y(i,1:2)');
        x_traj(i,:) = pos';
    end

    figure('Name', 'Controllo nello spazio operativo');
    plot(t, x_traj(:,1), 'b', 'LineWidth', 2);
    hold on;
    plot(t, x_traj(:,2), 'r', 'LineWidth', 2);
    plot([0 T_sim], [x_des x_des], 'b--');
    plot([0 T_sim], [y_des y_des], 'r--');
    grid on;
    legend('x', 'y', 'x_{target}', 'y_{target}');
    xlabel('Tempo [s]');
    ylabel('Posizione [m]');
    title('Convergenza alle coordinate cartesiane');
end

function plot_res_traj(t, q, dq, q_des, dq_des)
    % Calcolo dell'errore
    e_q = q_des - q;
    e_dq = dq_des -dq;

    figure('Name', 'Inseguimento traiettoria giunti');

    % Posizioni
    subplot(1,2,1);
    plot(t, q(:,1), 'b', 'LineWidth', 2);
    hold on;
    plot(t, q_des(:,1), 'k-.', 'LineWidth', 1.5);
    plot(t, q(:,2), 'r', 'LineWidth', 2);
    plot(t, q_des(:,2), 'k-.', 'LineWidth', 1.5);
    grid on;
```

```
xlabel('Tempo [s]');
ylabel('Posizione [rad]');
legend('q1', 'q1_{ref}', 'q2', 'q2_{ref}');
title('Inseguimento di posizione');

% Velocit
subplot(1,2,2);
plot(t, dq(:,1), 'b', 'LineWidth', 2);
hold on;
plot(t, dq_des(:,1), 'k-.', 'LineWidth', 1.5);
plot(t, dq(:,2), 'r', 'LineWidth', 2);
plot(t, dq_des(:,2), 'k-.', 'LineWidth', 1.5);
grid on;
xlabel('Tempo [s]');
ylabel('Velocit  [rad/s]');
legend('dq1', 'dq2');
title('Inseguimento di velocit ');

% Errore posizione
figure('Name', 'Errori di inseguimento');
subplot(1,2,1);
plot(t, e_q(:,1), 'b', 'LineWidth', 2);
hold on;
plot(t, e_q(:,2), 'r', 'LineWidth', 2);
grid on;
ylabel('Errore [rad]');
xlabel('Tempo [s]');
legend('e_{q1}', 'e_{q2}');
title('Errore in posizione');

% Errore velocit
subplot(1,2,2);
plot(t, e_dq(:,1), 'b', 'LineWidth', 2);
hold on;
plot(t, e_dq(:,2), 'r', 'LineWidth', 2);
```

```
    grid on;
    ylabel('Errore [rad/s]');
    xlabel('Tempo [s]');
    legend('e_{dq1}', 'e_{dq2}');
    title('Errore in velocit ');

end

function plot_res_op_traj(t, x_real, x_des, e_x)
    graphics_toolkit("gnuplot");

    figure('Name', 'Inseguimento traiettoria nello spazio operativo');

    % Traiettoria
    subplot(1,2,1);
    plot(x_real(:,1), x_real(:,2), 'g', 'LineWidth', 2);
    hold on;
    plot(x_des(:,1), x_des(:,2), 'k--', 'LineWidth', 2);
    axis equal;
    grid on;
    legend('Reale', 'Riferimento');
    xlabel('X [m]'); ylabel('Y [m]');
    title('Traiettoria cartesiana (cerchio)');

    % Errori
    subplot(1,2,2);
    plot(t, e_x(:,1), 'b', 'LineWidth', 2);
    hold on;
    plot(t, e_x(:,2), 'r', 'LineWidth', 2);
    grid on; ylabel('Errore [m]');
    xlabel('Tempo [s]');
    legend('e_x', 'e_y');
    title('Errore cartesiano');
end

function plot_res_adaptive(t, q, dq, theta_hist, q_des, theta_real)
    graphics_toolkit("gnuplot"); # fix per i grafici
```

```
% Calcolo errore
e_q = q_des - q;

figure('Name', 'Controllo adattativo');

% Errore di inseguimento
subplot(1,2,1);
plot(t, e_q(:,1), 'b', 'LineWidth', 1.5);
hold on;
plot(t, e_q(:,2), 'r', 'LineWidth', 1.5);
grid on;
ylabel('Errore [rad]');
xlabel('Tempo [s]');
legend('e_{q1}', 'e_{q2}');
title('Errore di inseguimento');

% Stima dei parametri
subplot(1,2,2);
colors = lines(8);
labels = {'\theta_1', '\theta_2', '\theta_3', '\theta_4', '\
    theta_5', '\theta_6', 'F_1', 'F_2'};

% Plot parametri stimati vs reali
hold on;
t_start = t(1);
t_end = t(end);
for i=1:8
    plot(t, theta_hist(:,i), 'Color', colors(i,:), 'LineWidth',
        1.5, 'DisplayName', ['Est. ' labels{i}]);
    plot([t_start t_end], [theta_real(i) theta_real(i)], '--', '
        Color', colors(i,:), 'LineWidth', 1, 'HandleVisibility', '
        off');
end
grid on; ylabel('Valore parametro');
```

```
    xlabel('Tempo [s]');
    title('Convergenza parametri (\theta)');
    legend('show', 'Location', 'eastoutside');
end
```

8 Robot uniciclo

Inseguimento di traiettoria in posizione per un robot uniciclo.

8.1 Modello cinematico

Il moto del robot uniciclo è un sistema non lineare descritto dalle seguenti equazioni cinematiche:

$$\begin{cases} \dot{x} = v\cos\theta \\ \dot{y} = v\sin\theta \\ \dot{\theta} = \omega \end{cases} \tag{82}$$

Dove:

- (x, y) è la posizione cartesiana nel piano.
- θ è l'orientamento rispetto all'asse x.
- v è la velocità lineare (ingresso di controllo).
- ω è la velocità angolare (ingresso di controllo).

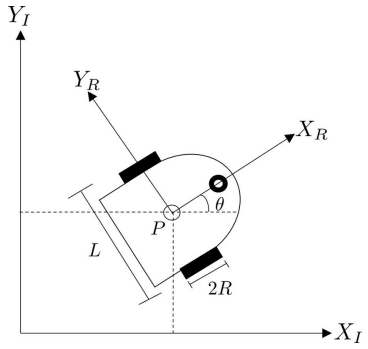

Figura 20: Robot uniciclo

8.2 Definizione dell'errore di inseguimento

8.2.1 Errore

Vogliamo che il robot segua un riferimento descritto da (x_d, y_d, θ_d) con ingressi di riferimento (v_d, ω_d). Prendo una traiettoria tipica di questi esempi: l'otto:

$$\begin{cases} x_d = A\sin(ft) \\ y_d = A\frac{\sin(2ft)}{2} \end{cases} \tag{83}$$

A differenza dei sistemi lineari, l'errore non è semplicemente $x_d - x$. Per i veicoli non olonomi, è necessario definire l'errore nel sistema di riferimento solidale al robot (robot). Usiamo la matrice di rotazione:

$$\begin{bmatrix} e_1 \\ e_2 \\ e_3 \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_d - x \\ y_d - y \\ \theta_d - \theta \end{bmatrix} \tag{84}$$

Dove:

- e_1: errore lungo x.
- e_2: errore lungo y.
- e_3: errore di orientamento rispetto a θ.

8.2.2 Dinamica dell'errore

Derivando le equazioni sopra e sostituendo il modello cinematico, otteniamo il sistema non lineare dell'errore:

$$\begin{cases} \dot{e}_1 = v_d \cos e_3 - v + e_2\omega \\ \dot{e}_2 = v_d \sin e_3 - e_1\omega \\ \dot{e}_3 = \omega_d - \omega \end{cases} \tag{85}$$

Questo è il sistema che dobbiamo stabilizzare all'origine $(0, 0, 0)$ usando gli ingressi (v, ω).

8.3 Progetto del controllore via Lyapunov

Il **secondo criterio di Lyapunov** afferma che sia 0 un punto di equilibrio per i sistema e $D \subset \mathbb{R}^n$ un dominio contenente l'origine. Sia $V : D \to \mathbb{R}$ con derivata continua su D. Assumendo che:

$$\begin{cases} V(0) = 0 \ \wedge \ V(x) > 0 \quad \forall x \in D \backslash \{0\} \\ \dot{V}(x) \leq 0 \quad \forall x \in D \end{cases} \tag{86}$$

allora x=0 è un punto di **equilibrio stabile**. Se $\dot{V}(x) < 0 \ \forall x \in D$ allora $x = 0$ è un punto di **equilibrio asintoticamente stabile**. Cerco, quindi, una funzione scalare $V(e)$ definita positiva e con derivata definita negativa. Una delle soluzioni più accreditate per esercizi simili è:

$$V = \frac{1}{2}e_1^2 + \frac{1}{2}e_2^2 + \frac{1}{k_2}(1 - \cos e_3) \tag{87}$$

con $k_2 > 0$. Questa funzione è definita positiva (nulla solo nell'origine e positiva altrove per $e_3 \in (-\pi, \pi)$).

8.3.1 Derivata $\dot{V}$

Calcoliamo la derivata temporale lungo le traiettorie del sistema e scegliamo gli ingressi per renderla negativa.

$$\dot{V} = e_1\dot{e}_1 + e_2\dot{e}_2 + \frac{1}{k_2}\sin e_3\dot{e}_3 \tag{88}$$

Sostituendo le equazioni della dinamica dell'errore e semplificando i termini incrociati ($e_1e_2\omega$), otteniamo l'espressione di $\dot{V}$:

$$\dot{V} = e_1(v_d\cos e_3 - v + e_2\omega) + e_2(v_d\sin e_3 - e_1\omega) + \frac{\sin e_3}{k_2}(\omega_d - \omega) \tag{89}$$

Semplificando i termini opposti (e_1 e_2 ω si annullano):

$$\dot{V} = e_1(v_d\cos e_3 - v) + v_de_2\sin e_3 + \frac{\sin e_3}{k_2}(\omega_d - \omega) \tag{90}$$

8.3.2 Scelta della legge di controllo

Per rendere $\dot{V} \leq 0$, scegliamo v e ω in modo da annullare i termini più ostici e iniettare smorzamento. Per il primo termine, poniamo:

$$v_d\cos e_3 - v = -k_1e_1 \implies v = v_d\cos e_3 + k_1e_1 \tag{91}$$

Questo rende il primo termine pari a $-k_1e_1^2$ (definito negativo). Per i restanti termini, raccogliamo $\sin e_3$:

$$\sin e_3\left(v_de_2 + \frac{\omega_d - \omega}{k_2}\right) \tag{92}$$

Imponiamo che la parentesi sia uguale a un termine dissipativo, ad esempio $-\frac{k_3}{k_2}\sin e_3$:

$$v_de_2 + \frac{\omega_d - \omega}{k_2} = -\frac{k_3}{k_2}\sin e_3 \tag{93}$$

Risolvendo per $\omega = \omega_d + k_2v_de_2 + k_3\sin e_3$. Ora posso scrivere il controllo come:

$$\begin{cases} v = v_d\cos e_3 + k_1e_1 \\ \omega = \omega_d + k_2v_de_2 + k_3\sin e_3 \end{cases} \tag{94}$$

Il guadagno k_1 agisce principalmente sulla convergenza dell'errore lungo la direzione di moto x mentre k_2 e k_3 influenzano la dinamica laterale y e di orientamento θ. Un valore elevato di k_2 permette di correggere rapidamente l'errore laterale e_2, ma può indurre oscillazioni se eccessivo.

8.3.3 Verifica della stabilità

Sostituendo questi ingressi nella derivata di Lyapunov, si ottiene:

$$\dot{V} = -k_1 e_1^2 - \frac{k_3}{k_2} \sin^2 e_3 \leq 0 \tag{95}$$

La derivata è semidefinita negativa. Utilizzando il Lemma di Barbalat (e verificando l'uniforme continuità dei segnali), si può dimostrare la convergenza asintotica dell'errore a zero, a patto che la velocità di riferimento v_d non sia costantemente nulla (condizione di eccitazione).

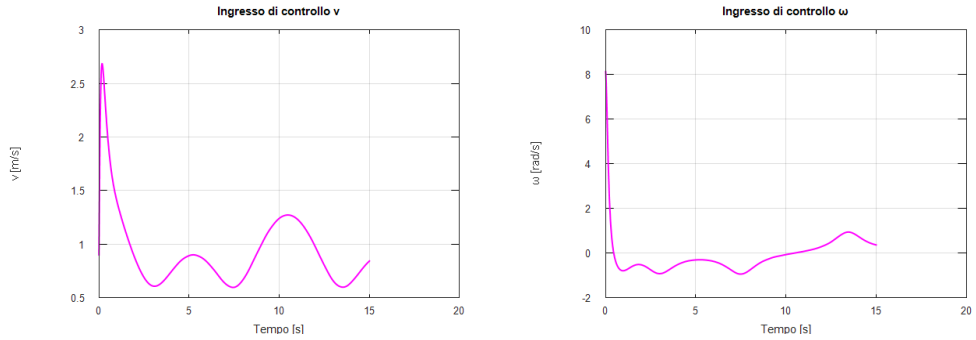

Figura 21: Segnali di controllo v (sinistra) e ω (destra)

Gli ingressi v e ω rimangono limitati e continui come si vede in Figura 21. Questo è fondamentale per la realizzabilità fisica del controllo su attuatori reali (motori). La velocità angolare ω presenta delle variazioni più marcate nelle curve strette dell'otto, come previsto dal termine di feedforward ω_d.

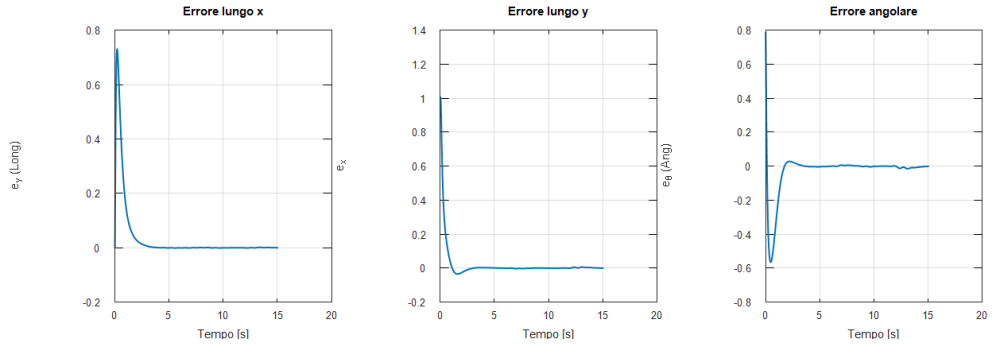

Figura 22: Andamento degli errori di inseguimento

Gli errori e_x, e_y ed e_θ tendono asintoticamente a zero come si nota in Figura 22. Si osservano delle piccole oscillazioni durante i cambi di curvatura della traiettoria, che vengono prontamente compensate dai termini di feedback.

Il robot converge rapidamente sulla traiettoria di riferimento a forma di otto in Figura 23. L'errore iniziale di posizione ($y = -1$) viene corretto nei primi secondi di simulazione grazie all'azione combinata di v e ω.

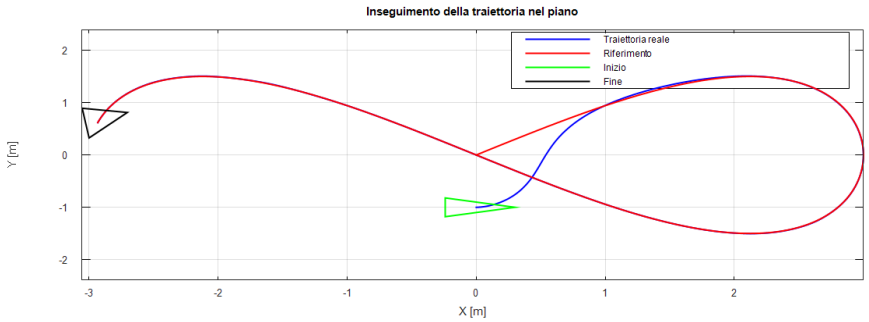

Figura 23: Traiettoria desiderata e reale del robot uniciclo

8.4 Codice

Lo script principale `unicycle_robot.m` si occupa della configurazione dei parametri, dell'inizializzazione e del loop di simulazione.

```
clear all;
close all;
clc;

addpath('8')

% Parametri del controllore
k1 = 2.0;
k2 = 5.0;
k3 = 2.5;

global gains
gains = [k1, k2, k3];

% Configurazione della simulazione
T_sim = 15;          % Durata [s]
dt = 0.01;           % Passo
t_span = 0:dt:T_sim;

% Condizioni iniziali del robot (x, y, theta)
q0 = [0; -1; 0];
```

```
% Simulazione
disp('Avvio simulazione del robot uniciclo');
[t, q] = ode45(@(t,q) unicycle_dynamics(t, q), t_span, q0);

% Ricostruiamo i segnali di riferimento e di errore per i grafici
x = q(:,1);
y = q(:,2);
theta = q(:,3);
N = length(t);
xd = zeros(N,1);
yd = zeros(N,1);
thetad = zeros(N,1);
e1 = zeros(N,1);
e2 = zeros(N,1);
e3 = zeros(N,1);
v_cmd = zeros(N,1);
w_cmd = zeros(N,1);

for i=1:N
    [dqdt, ref, err, ctrl] = unicycle_dynamics(t(i), q(i,:)');
    xd(i) = ref(1);
    yd(i) = ref(2);
    thetad(i) = ref(3);
    e1(i) = err(1);
    e2(i) = err(2);
    e3(i) = err(3);
    v_cmd(i) = ctrl(1);
    w_cmd(i) = ctrl(2);
end

% Plot delle traiettorie degli errori e dei riferimenti
plot_res(x, y, xd, yd, e1, e2, e3, v_cmd, w_cmd, t, theta)
```

La funzione `unicycle_dynamics.m` implementa le equazioni cinematiche, genera il riferimento e calcola la legge di controllo.

```
function [dqdt, ref, err, ctrl] = unicycle_dynamics(t, q)
```

```
global gains
k1 = gains(1); k2 = gains(2); k3 = gains(3);

x = q(1); y = q(2); theta = q(3);

% Traiettoria a 8
A = 3;
w0 = 0.3; % frequenza base
xd = A * sin(w0*t);
yd = A * sin(2*w0*t) / 2;

% Derivate prime (velocit  cartesiane riferimento)
dxd = A * w0 * cos(w0*t);
dyd = A * 2 * w0 * cos(2*w0*t) / 2;

% Derivate seconde (accelerazioni cartesiane riferimento)
ddxd = -A * w0^2 * sin(w0*t);
ddyd = -A * 4 * w0^2 * sin(2*w0*t) / 2;

% Calcolo riferimenti non-olonomi (vd, wd, thetad)
vd = sqrt(dxd^2 + dyd^2);
thetad = atan2(dyd, dxd);
wd = (dxd*ddyd - dyd*ddxd) / (vd^2 + 1e-6);

% Calcolo errori nel body frame
ex_world = xd - x;
ey_world = yd - y;
e_theta = thetad - theta;

% Normalizzazione angolo tra -pi e pi
e_theta = atan2(sin(e_theta), cos(e_theta));
e1 =  cos(theta)*ex_world + sin(theta)*ey_world;
e2 = -sin(theta)*ex_world + cos(theta)*ey_world;
e3 = e_theta;
```

```
    % Legge di controllo
    v = vd * cos(e3) + k1 * e1;
    w = wd + k2 * vd * e2 + k3 * sin(e3);

    % 4. Dinamica del robot reale
    dxdt = v * cos(theta);
    dydt = v * sin(theta);
    dthetadt = w;
    dqdt = [dxdt; dydt; dthetadt];

    % Output extra per post-processing
    ref = [xd; yd; thetad];
    err = [e1; e2; e3];
    ctrl = [v; w];
end
```

La funzione `plot_res.m` permette di visualizzare l'azione di controllo, l'evoluzione degli errori rispetto ai riferimenti e la traiettoria del robot rispetto a quella desiderata.

```
function plot_res(x, y, xd, yd, e1, e2, e3, v_cmd, w_cmd, t, theta)
    graphics_toolkit("gnuplot"); # fix per i grafici

    % Traiettoria nel piano XY
    figure(1);
    plot(x, y, 'b', 'LineWidth', 2);
    hold on;
    plot(xd, yd, 'r', 'LineWidth', 2);

    % Disegna il robot all'inizio e alla fine
    draw_robot(x(1), y(1), theta(1), 'g');
    draw_robot(x(end), y(end), theta(end), 'k');
    grid on;
    axis equal;
    title('Inseguimento della traiettoria nel piano');
    legend('Traiettoria reale', 'Riferimento', 'Inizio', 'Fine');
    xlabel('X [m]');
    ylabel('Y [m]');
```

```
% Errori nel tempo
figure(2);
subplot(1,3,1);
plot(t, e1, 'LineWidth', 2);
grid on;
ylabel('e_y (Long)');
xlabel('Tempo [s]');
title('Errore lungo x');

subplot(1,3,2);
plot(t, e2, 'LineWidth', 2);
grid on;
ylabel('e_x');
xlabel('Tempo [s]');
title('Errore lungo y');

subplot(1,3,3);
plot(t, e3, 'LineWidth', 2);
grid on;
ylabel('e_\theta (Ang)');
xlabel('Tempo [s]');
title('Errore angolare');

% Ingressi di controllo
figure(3);
subplot(1,2,1);
plot(t, v_cmd, 'm', 'LineWidth', 2);
grid on;
xlabel('Tempo [s]');
ylabel('v [m/s]');
title('Ingresso di controllo v');

subplot(1,2,2);
plot(t, w_cmd, 'm', 'LineWidth', 2);
```

```
    grid on;
    ylabel('\omega [rad/s]');
    xlabel('Tempo [s]');
    title('Ingresso di controllo \omega');
end
```

Ho scritto la funzione `draw_robot.m` per identificare l'orientamento del robot nella raffigurazione della traiettoria.

```
function draw_robot(x, y, th, col)
    % Disegna un semplice triangolo per rappresentare il robot
    R = 0.3; % Dimensione robot
    p1 = [x + R*cos(th); y + R*sin(th)];
    p2 = [x + R*cos(th + 2.5); y + R*sin(th + 2.5)];
    p3 = [x + R*cos(th - 2.5); y + R*sin(th - 2.5)];
    plot([p1(1) p2(1) p3(1) p1(1)], [p1(2) p2(2) p3(2) p1(2)], col, '
        LineWidth', 2);
end
```

9 Funzioni ausiliarie

La funzione `is_pos_definite` verifica se una matrice è definita positiva controllando che tutti i suoi autovalori siano strettamente maggiori di una tolleranza positiva.

```
function controllable = is_controllable(A, B)
    % Costruisco manualmente la matrice di controllabilit : R_AB = [B
       , AB, A^2B, ..., A^(n-1)B]
    R_AB = B;
    A_power = A;
    [n, ~] = size(A);

    % Aggiungo iterativamente le colonne AB, A^2B, ..., A^(n-1)B
    for i = 1:(n-1)
        R_AB = [R_AB, A_power * B];
        A_power = A_power * A;
    end

    % Controllo del rango pieno
    rank_R_AB = rank(R_AB);
    if rank_R_AB < n
        controllable = false;
    else
        controllable = true;
    end
end
```

La funzione `is_pos_semidefinite` verifica se una matrice è semidefinita positiva controllando che tutti i suoi autovalori siano maggiori o uguali a una tolleranza.

```
function detectable = is_detectable(C, A, L)
    % Verifico lo spettro della matrice (A-LC)
    tol=1e-6
    det_poles= eig(A - L*C);
    if max(real(det_poles)) < tol
        detectable = true;
    else
        detectable = false;
```

```
    end
end
```

La funzione `is_controllable` verifica la controllabilità della coppia (A, B) costruendo la matrice di controllabilità $\mathcal{R}_{AB}$ e controllando che abbia rango pieno (n).

```
function observable = is_observable(A, C)
    [n, ~] = size(A);

    % Costruisco manualmente la matrice di osservabilit
    O_AC = C;
    A_power = A;

    % Aggiungo iterativamente le righe C*A, C*A^2, ...
    for i = 1:(n-1)
        O_AC = [O_AC; C * A_power];
        A_power = A_power * A;
    end

    % Controllo del rango pieno
    rank_O_AC = rank(O_AC);
    if rank_O_AC < n
        observable = false;
    else
        observable = true;
    end
end
```

La funzione `is_observable` verifica l'osservabilità della coppia (A, C) costruendo la matrice di osservabilità $\mathcal{O}_{AC}$ e controllando che abbia rango pieno (n).

```
function pos_def = is_pos_definite(A)
    % Controlla se la matrice    definita posistiva
    tol = 1e-6
    eigenvalues = eig(A);
    pos_def = all(eigenvalues > tol);
end
```

La funzione `is_stabilizable` verifica se il sistema a ciclo chiuso con guadagno K è stabile, controllando che gli autovalori della matrice dinamica $(A - BK)$ abbiano parte reale negativa.

```
function pos_semidef = is_pos_semidefinite(A)
    % Controlla se la matrice    semidefinita posistiva
    tol = -1e-6
    eigenvalues = eig(A);
    pos_semidef = all(eigenvalues >= tol);
end
```

La funzione `is_detectable` verifica se l'osservatore con guadagno L è stabile, controllando che gli autovalori della matrice dell'errore $(A - LC)$ abbiano parte reale negativa.

```
function stabilizable = is_stabilizable(A, B, K)
    % Verifico lo spettro della matrice (A-BK)
    tol = 1e-6
    stab_poles= eig(A - B*K);
    if max(real(stab_poles)) < tol
        stabilizable = true;
    else
        stabilizable = false;
    end
end
```