# Progetto ISI

Jacopo Conti Matteo Gafforio

Anno Accademico 2023/2024

# Indice

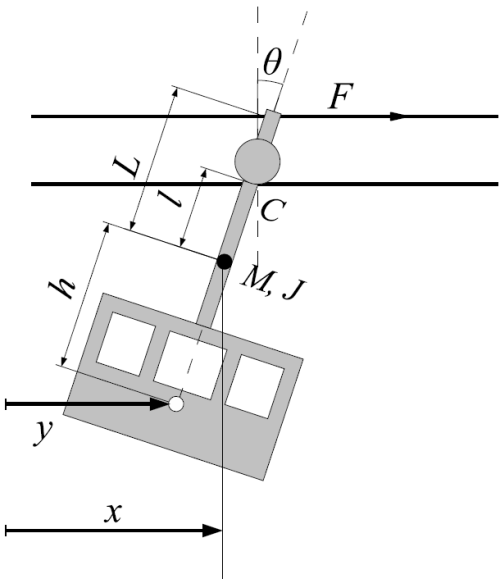



Figura 1: funivia

# 1 Introduzione

## 1.1 Descrizione del modello

Per il nostro progetto abbiamo scelto la **traccia C**, ovvero la funivia mostrata in figura 1:

La cabina è sorretta da un cavo di sostegno, tramite una puleggia, e il movimento lungo x è guidato da un cavo di traino superiore che esercita una forza F. Nel centro della puleggia agisce inoltre una forza modellata come attrito viscoso.

Il sistema ha quattro variabili di stato, ovvero:

$$x;\ \theta;\ \dot{x};\ \ddot{\theta}; \tag{1}$$

dove x rappresenta la distanza del centro di massa dall'origine e $\theta$ l'angolo di inclinazione rispetto alla verticale.

## 1.2 Descrizione sensori assegnati

Per il progetto dello stimatore avevamo a disposizione tre sensori in grado di misurare tre grandezze:

- **sensore 1:** misura la distanza D
- **sensore 2:** misura la velocità angolare della puleggia di sostegno omega
- **sensore 3:** misura la distanza d2

Queste grandezze sono visibili in figura 2.

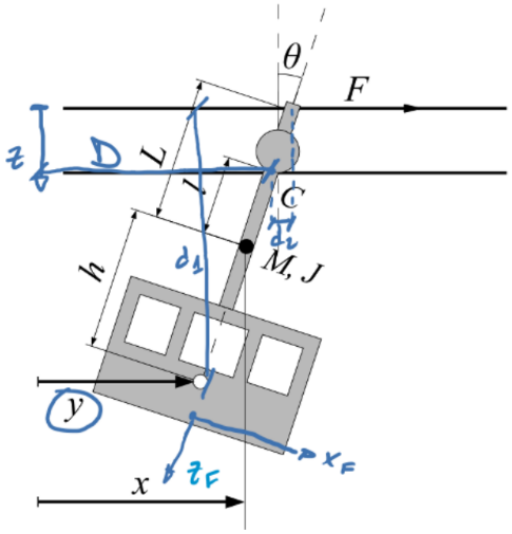



Figura 2: funivia con grandezze misurabili

# 2 Implementazione e validazione Modello

## 2.1 Analisi e scelta dei parametri nominali del sistema

Il modello dinamico associato al sistema è descritto dalle seguenti equazioni:

$$M\ddot{x} = -c_a v_c + F \tag{2}$$

$$J\ddot{\theta} = -Mgl\sin\theta + c_a v_c l\cos\theta - FL\cos\theta \tag{3}$$

dove:

$$v_c = \dot{x} - l\dot{\theta}\cos\theta \tag{4}$$

Tuttavia, avendo a disposizione un sensore per misurare la velocità angolare della puleggia di sostegno, e usando il raggio di questa per il calcolo della velocità lineare, abbiamo deciso di non trascurarla. Quindi la (2) rimane invariata, mentre la (3) e la (4) diventano:

$$J\ddot{\theta} = -Mg(l+r)\sin\theta + c_a v_c(l+r)\cos\theta - FL\cos\theta \tag{5}$$

$$v_c = \dot{x} - (l+r)\dot{\theta}\cos\theta \tag{6}$$

Come si può notare il raggio della puleggia va a sommarsi nella (5) alla lunghezza l sia nel termine della reazione vincolare sia in quello dell'attrito. Mentre nella (6) si somma al termine che tiene di conto della distanza tra il centro di massa e la puleggia

**Scelta parametri nominali.** Abbiamo effettuato le seguenti scelte che riteniamo ragionevoli per i parametri nominali:

$$M = 2000[kg];\ J = 2800[Kg * m^2];\ c_a = 20[N * s/m];\ L = 3[m];\ h = 2.25[m];\ r = 0.1[m]; \tag{7}$$

Per quanto riguarda la massa abbiamo utilizzato un dato simile[1] a quello trovato qui, considerando una media di 10 persone caricate sulla cabina. Il momento d'inerzia invece è un valore simile a quello ottenuto considerando tutta la massa concentrata nel centro di massa. Le lunghezze sono valori che sono ragionevoli per avere giuste proporzioni rispetto alla dimensione della cabina.

[1] considerando che si tratta di un impianto molto vecchio, abbiamo supposto un peso a vuoto superiore di circa il 30% per una cabina moderna di dimensioni quasi doppie

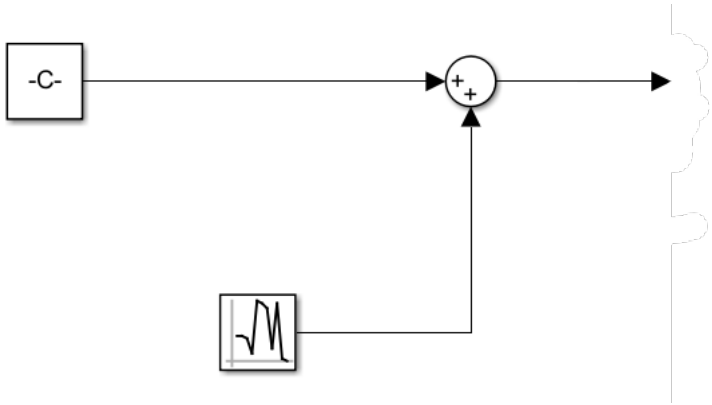



Figura 3: Modello Simulink del segnale di ingresso

## 2.2 Implementazione

Il modello dinamico è stato implementato in ambiente Simulink, attraverso una Matlab function, con il seguente codice:

Listing 1: Codice modello

```
1
2     function [q_dd,Vc,z] = fcn(F,q,q_d,param)
3
4     M = param.M;
5     J = param.J;
6     Ca = param.Ca;
7     L = param.L;
8     l = param.l;
9     h = param.h;
10    g = param.g;
11    r = param.r;
12
13    z = L*cos(q(2));
14
15    Vc = q_d(1) - (l+r)*q_d(2)*cos(q(2));
16    %velocità del punto C di contatto tra
17    %la puleggia ed il cavo di sostegno
18
19    x_dd = (-Ca*Vc + F)/M;
20    %prima equazione cardinale
21
22    tetha_dd = (-M*g*(l+r)*sin(q(2)) +
23                Ca*(l+r)*Vc*cos(q(2)) - F*L*cos(q(2)))/J;
24    %seconda cardinale scritta rispetto al baricentro
25
26    q_dd = [x_dd; tetha_dd];
```

L'unico ingresso al sistema (figura 3) è la Forza F, che sposta la funivia in direzione orizzontale. L'incertezza, sul valore di questa forza, è modellata come un rumore bianco gaussiano aggiunto in modo additivo.

All'esterno della MatLab function, ci sono due integratori in retroazione per ricavare gli ingressi alla funzione. Al loro interno vengono definite anche le condizioni iniziali del sistema.

In uscita dal blocco infine abbiamo la q e $v_c$, questi vanno in ingresso ai blocchi che contengono i modelli dei sensori. (La struttura appena descritta si può osservare in figura 4).

## 2.3 Validazione Modello

Per la validazione del modello, inizialmente abbiamo scollegato i rumori dell'ingresso e abbiamo fatto le seguenti prove:

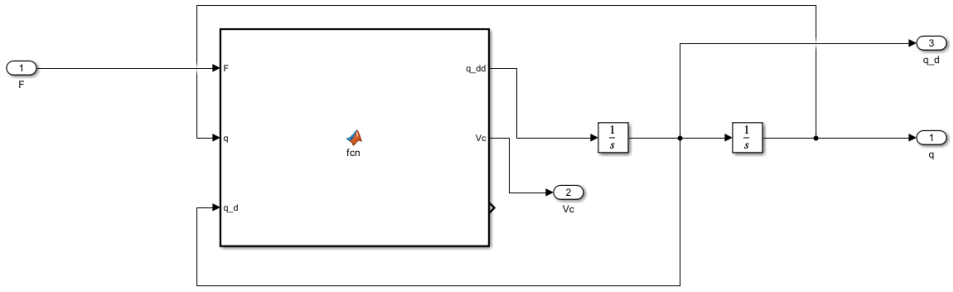



Figura 4: Schema a blocchi del modello nominale

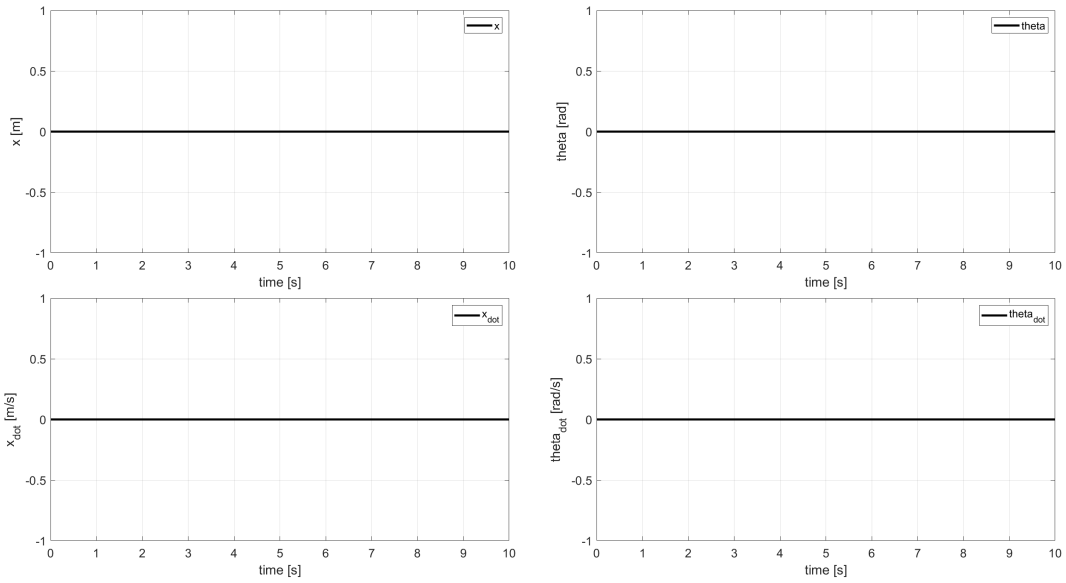



Figura 5: $F = 0[N]$

1. Abbiamo inizializzato la funivia con inclinazione nulla e applicare ingresso nullo. Ci aspettavamo che in queste condizioni la funivia rimanesse ferma, in figura 5 possiamo vedere i risultati.

2. Successivamente abbiamo impostato l'inclinazione iniziale a 15°. In questo caso a causa dell'attrito ci aspettavamo che la funivia si arrestasse e i risultati sono in accordo con le nostre aspettative (vedi figura 6)(Per velocizzare il processo abbiamo aumentato il valore del coefficiente di attrito).

3. Un ulteriore prova è stata impostare la forza $F = 500N$. Chiaramente ci aspettavamo che la funivia si spostasse in avanti, con la velocità che tendesse ad aumentare fino ad assestarsi ad un certo valore (quando arriviamo alla condizione di bilanciamento delle forze). I risultati in figura 7. Questa prova l'abbiamo ripetuta anche scambiando il segno alla forza e alle condizioni iniziali, aspettandoci dei risultati simmetrici, esattamente come in figura 8.

4. Come ultima verifica, abbiamo provato a portare la funivia alla velocità di regime che abbiamo ipotizzato a $10[m/s]$. Per far ciò abbiamo calcolato la forza F necessaria (il risultato ottenuto è stato $F = v_c c_a = 200[N]$) e i risultati (figura 9) confermano ciò che avevamo ipotizzato.

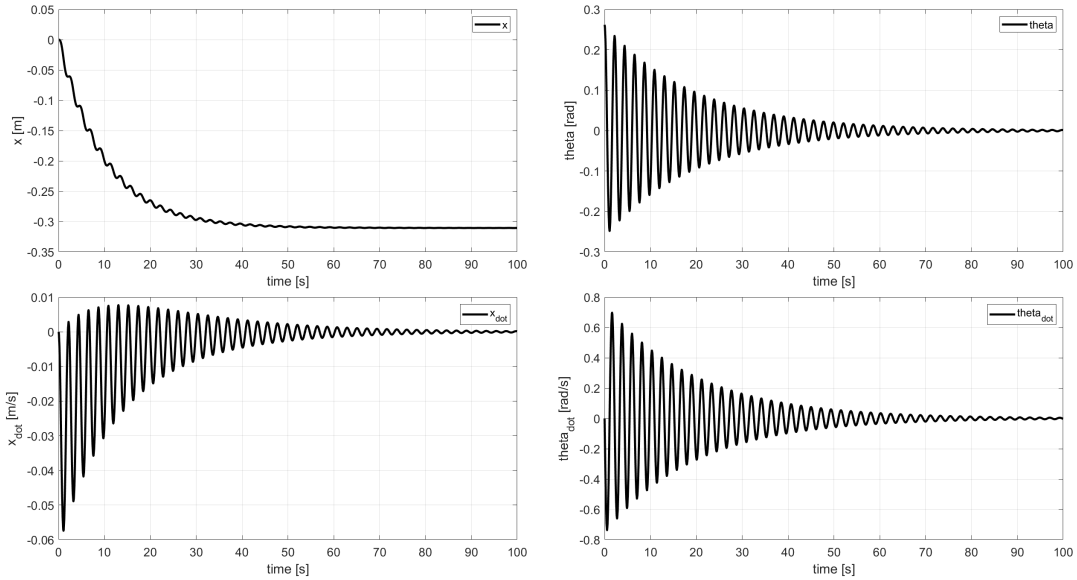



Figura 6: $F = 0[N]$ $\theta = \frac{\pi}{12}[rad]$

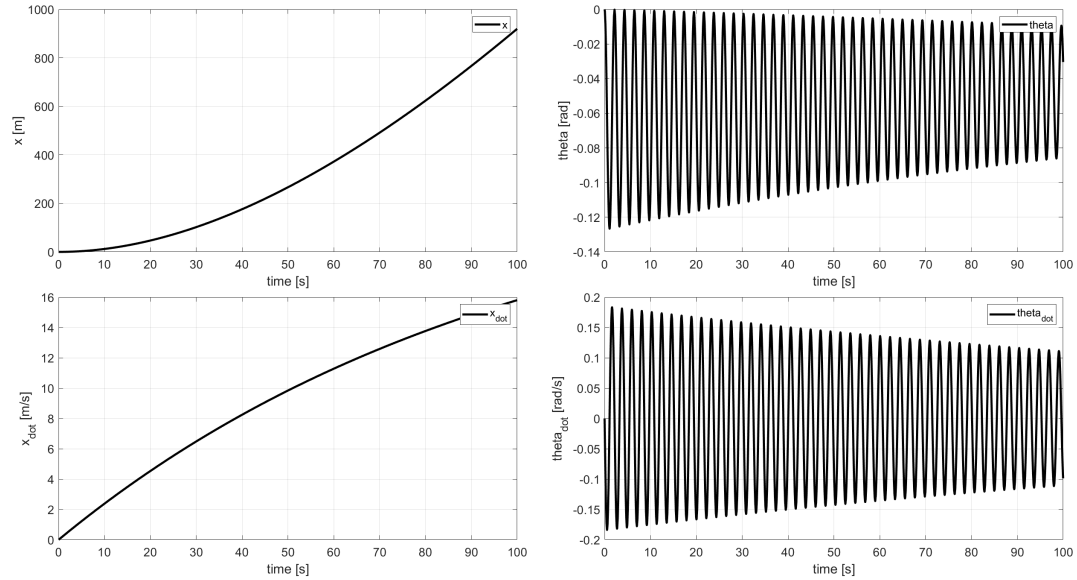



Figura 7: $F = 500[N]$

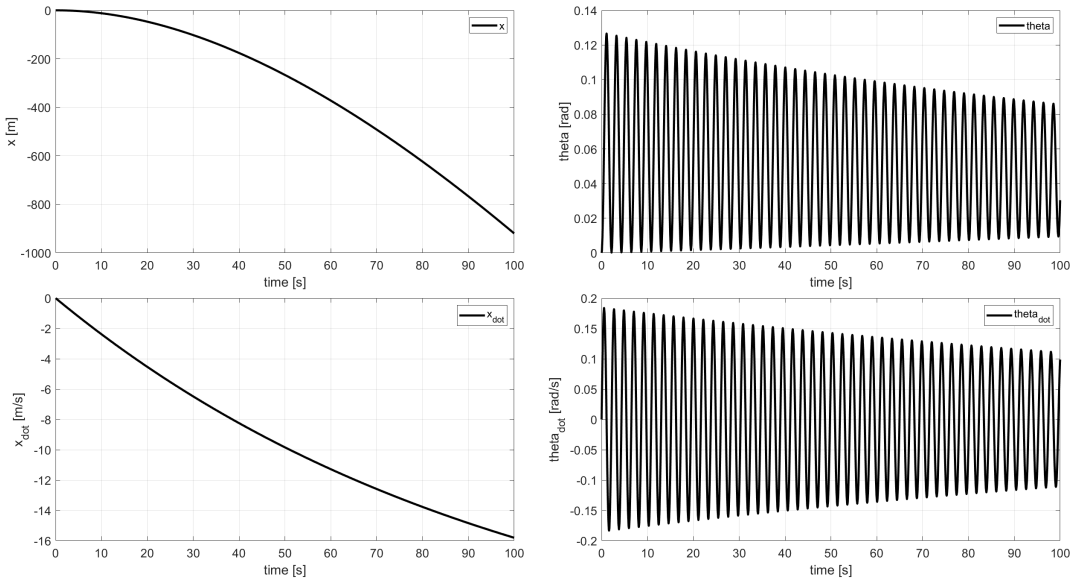



Figura 8: $F = -500[N]$

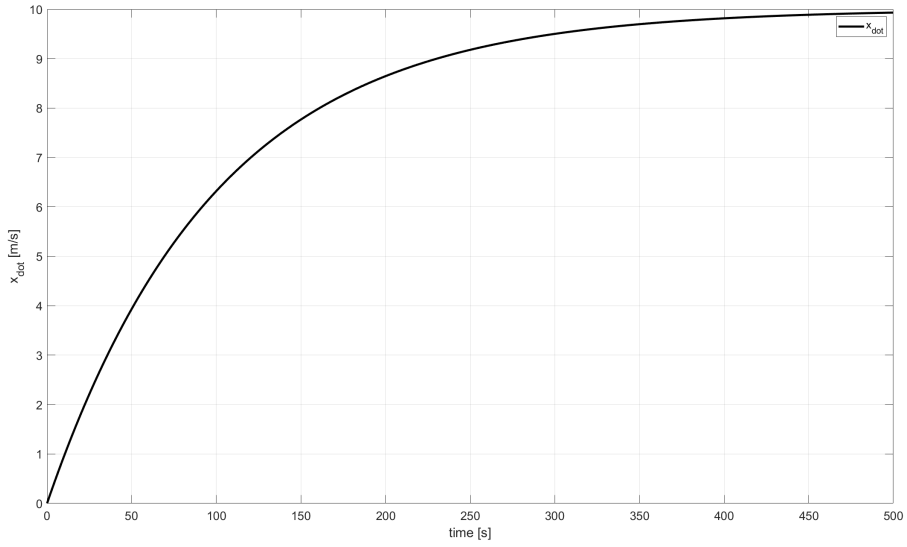



Figura 9: $F = 200[N]$

# 3 Implementazione dei Filtri

## 3.1 Scelta sensori

Per la scelta dei sensori da applicare al sistema, abbiamo cercato sistemi reali simili a quello che avevamo davanti. Da questi abbiamo ricavato delle ipotesi sulle caratteristiche dei sensori che avremmo dovuto applicare. Successivamente abbiamo trovato dispositivi che rispecchiassero questi parametri. I sensori trovati sono:

- **Sensore per D.**Avevamo bisogno di un sensore che potesse misurare lunghe distanze, questi tipi di sensori sono più lenti rispetto agli altri due, abbiamo ritenuto che una frequenza di 10Hz fosse sufficiente per la nostra applicazione, abbiamo quindi cercato un sensore che lavorasse almeno a questa frequenza, con errore massimo inferiore a 2 metri. Il sensore scelto è di tipo laser (link sensore).
- **Sensore per d2.**In questo caso c'era necessità di maggiore accuratezza rispetto a Dc, senza necessitare di un grande range[2], abbiamo scelto un sensore a ultrasuoni (link sensore).
- **Sensore per omega.**La scelta è stata un encoder ottico incrementale (link sensore). Abbiamo calcolato la massima velocità di rotazione per vedere se questo sensore fosse effettivamente utilizzabile nel nostro sistema:

$$\omega_{max} = \frac{v_{c_{max}}}{r} = \frac{10}{0.1} = 100Hz^3 \tag{8}$$

Per aggiungere i rumori alle misure, abbiamo utilizzato il solito metodo usato per l'ingresso F. Successivamente, per sporcare la stima dei filtri, data la troppa precisione dei sensori, abbiamo diminuito la precisione di tutti i sensori fino ad ottenere dei risultati in cui fosse apprezzabile l'errore di stima dello stato.

Listing 2: Codice implementazione caratteristica sensori

```
% sensore Dc (radio)
sensor.Dc.freq = 10;                                %[Hz]
precisione_Dc = 1;                                  %[m]
sensor.Dc.std_dev = sqrt(precisione_Dc/3);          %[m]       %std_dev=precisone/3
sensor.Dc.seed = 19;                                %seed per generare i
                                                    %soliti rumori

% sensore encoder
sensor.omega.freq = 1000;                           %[Hz]
precisione_omega = 1;                               %[rad/s]
sensor.omega.std_dev = sqrt(precisione_omega/3);%[rad/s]  %std_dev=precisone/3
sensor.omega.seed = 255;                            %seed per generare i
                                                    %soliti rumori

% sensore d2
sensor.d2.freq = 1000;                              %[Hz]
precisione_d2 = 0.1;                                %[m]
sensor.d2.std_dev = sqrt(precisione_d2/3);          %[m]       %std_dev=precisone/3
sensor.d2.seed = 45;                                %seed per generare i
                                                    %soliti rumori
```

Abbiamo poi ricavato le formule delle misure ottenute, in funzione delle variabili di stato, da inserire all'interno dei filtri per eseguire la correzione:

$$Dc = x - (l + r)sin(\theta) \tag{9}$$

$$d2 = -(L - r - l)sin(\theta) \tag{10}$$

$$\omega = \dot{x}/r + (l + r)\dot{\theta}\cos\theta/r \tag{11}$$

[2]nel caso di funivia inclinata di 90°, che è chiaramente un caso assurdo, il sensore dovrebbe misurare 1.9m
[3]ipotizzando la velocità massima della funivia di 10m/s prendendo un dato simile a funivia 1 e funivia 2

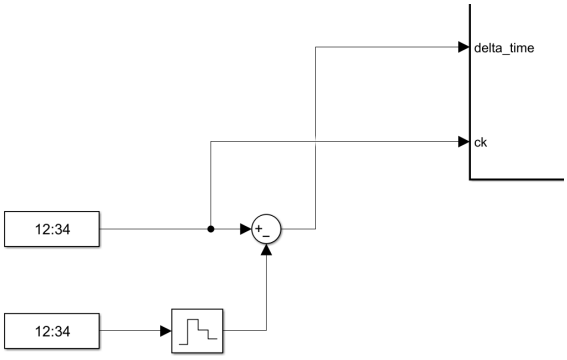



Figura 10: Generazione dei segnali di correzione

## 3.2 Scelta frequenza filtri

Per la scelta delle frequenze di funzionamento dei filtri, avendo la necessità di predire e correggere a frequenze diverse, abbiamo usato le frequenze del sensore più lento e più veloce per ricavarci questi dati. Abbiamo creato del codice che, date le frequenze dei 3 sensori, ci restituisse la minima e la massima (permettendoci di poter cambiare tipologia di sensori senza effettuare altri cambiamenti nel codice). Il codice che realizza ciò è il seguente:

Listing 3: Codice modello

```
%trovo le frequenze del sensore piu lento e piu veloce
if (sensor.omega.freq > sensor.Dc.freq)
    f_max = sensor.omega.freq;
    f_min = sensor.Dc.freq;
    if (sensor.d2.freq > f_max)
        f_max = sensor.d2.freq;
    elseif (sensor.d2.freq < f_min)
        f_min = sensor.d2.freq;
    end
else
    f_min = sensor.omega.freq;
    f_max = sensor.Dc.freq;
    if (sensor.d2.freq > f_max)
        f_max = sensor.d2.freq;
    elseif (sensor.d2.freq < f_min)
        f_min = sensor.d2.freq;
    end
end
```

A questo punto abbiamo scelto la frequenza di predizione uguale a quella del sensore più veloce. Per la correzione, abbiamo generato dei segnali che ci permettessero di capire quando poter correggere. In figura 10 possiamo vedere:

1. Il segnale **delta_time** generato dai tre blocchi Simulink come differenza tra il tempo conteggiato da un clock a frequenza massima e uno a frequenza minima.
2. Il segnale **ck**.

Poi tramite un questo semplice codice:

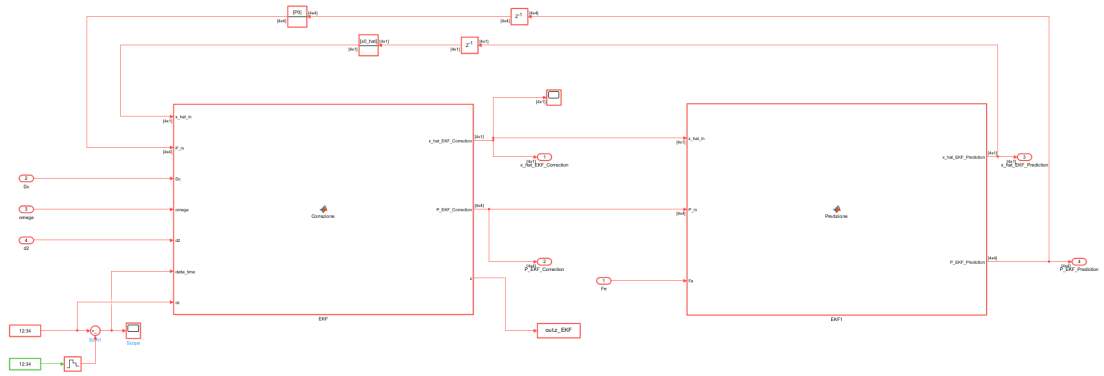
Listing 4: Codice modello

```
if (delta_time <= 0) && (ck ~= 0)
```

il filtro corregge solo nel caso abbia a disposizione dati nuovi di tutti i sensori.

## 3.3 EKF

Abbiamo implementato l'EKF in ambiente Simulink in questo modo:



Abbiamo utilizzato due MatLab function, una per la correzione ed una per la predizione. Abbiamo poi creato il loop inserendo all'interno due tempi di ritardo (uno per linea) per non generare un loop algebrico e due blocchi per le condizioni iniziali. In ingresso al filtro abbiamo le misure dei sensori e la forza applicata sul modello.

In particolare, il codice nel blocco di correzione è il seguente:

Listing 5: Codice correzione ekf

```
1
2       function [ x_hat_EKF_Correction, P_EKF_Correction, z]=
3       Correzione (x_hat_in, P_in, Dc, omega, d2, param,...
4                   sensor, Fext, delta_time, ck)
5
6      M = param.M;
7      J = param.J;
8      Ca = param.Ca;
9      L = param.L;
10     l = param.l;
11     h = param.h;
12     g = param.g;
13     r = param.r;
14     Dc_std_dev = sensor.Dc.std_dev;
15     d2_std_dev = sensor.d2.std_dev;
16     omega_std_dev = sensor.omega.std_dev;
17
18     x_hat_EKF_Correction = x_hat_in;
19
20     % P = [init.q_std_dev(1)^2 0 0 0;...     % incertezza iniziale
21     %       0 init.q_std_dev(2)^2 0 0;...
22     %       0 0 init.q_d_std_dev(1)^2 0;...
23     %       0 0 0 init.q_d_std_dev(2)^2];
24     P_EKF_Correction = P_in;
25
26     R = [Dc_std_dev^2 0 0;...                % matrice incertezze dei sensori
27          0 d2_std_dev^2 0;...
28          0 0 omega_std_dev^2];
29
30     %% Correzione
31     if (delta_time <= 0) && (ck ~= 0)        %algoritmo di scelta di correzione
32
33         e = zeros(3,1);
34         e(1) = Dc - ( x_hat_EKF_Correction(1) - ...
35                 (l+r)*sin(x_hat_EKF_Correction(2)) );
36         e(2) = d2 + ( (L-l-r)*sin(x_hat_EKF_Correction(2)) );
37         e(3) = omega - ( x_hat_EKF_Correction(3) -
               (l+r)*x_hat_EKF_Correction(4)*cos(x_hat_EKF_Correction(2)) )/r;
38
39         H = [1     -(l+r)*cos(x_hat_EKF_Correction(2))                    0
                                  0                                    ;...
40              0     -(L-l-r)*cos(x_hat_EKF_Correction(2))                  0
                                  0                                    ;...
41              0 ((l+r)*x_hat_EKF_Correction(4)*sin(x_hat_EKF_Correction(2)))/r  1/r
                    -(l+r)/r*cos(x_hat_EKF_Correction(2))];
42
```

```
43          %M = eye(3);
44          % dato che è l'identità non lo consideriamo nelle formule successive
45
46          S = H*P_EKF_Correction*H' + R;
47          Lk = P_EKF_Correction*H'/S;
48
49          x_hat_EKF_Correction = x_hat_EKF_Correction + Lk*e;
50          P_EKF_Correction = P_EKF_Correction - Lk*S*Lk';
51      end
52
53      z = L*cos(x_hat_EKF_Correction(2));      %variabile usato per il plot
```

Mentre il codice per la predizione è:

Listing 6: Codice predizione ekf

```
1       function [x_hat_EKF_Prediction, P_EKF_Prediction] = Predizione (x_hat_in,
            P_in, Fe, param, sensor, Fext, dT)
2
3       M = param.M;
4       J = param.J;
5       Ca = param.Ca;
6       L = param.L;
7       l = param.l;
8       h = param.h;
9       g = param.g;
10      r = param.r;
11
12
13      x_hat_EKF_Prediction = x_hat_in;
14
15      % P = [init.q_std_dev(1)^2 0 0 0;...          % incertezza iniziale
16      %       0 init.q_std_dev(2)^2 0 0;...
17      %       0 0 init.q_d_std_dev(1)^2 0;...
18      %       0 0 0 init.q_d_std_dev(2)^2];
19      P_EKF_Prediction = P_in;
20
21      Q = Fext.std_dev^2;                          % incertezza ingresso
22
23      %% Predizione
24      x_hat_EKF_Prediction(1) = x_hat_EKF_Prediction(1) + dT*x_hat_EKF_Prediction(3);
25      x_hat_EKF_Prediction(2) = x_hat_EKF_Prediction(2) + dT*x_hat_EKF_Prediction(4);
26      x_hat_EKF_Prediction(3) = (1 - dT*Ca/M)*x_hat_EKF_Prediction(3) +...
27                                (dT*Ca*(l+r)/M)*x_hat_EKF_Prediction(4)*...
28                                cos(x_hat_EKF_Prediction(2)) + (dT*Fe/M);
29      x_hat_EKF_Prediction(4) = (1 - dT*Ca*((l+r)^2)/J*...
30                                (cos(x_hat_EKF_Prediction(2))^2))...
31                                *x_hat_EKF_Prediction(4) - (dT*M*g*...
32                                (l+r)/J)*sin(x_hat_EKF_Prediction(2)) +...
33                                dT/J*(Ca*(l+r)*x_hat_EKF_Prediction(3) - ...
34                                Fe*L)*cos(x_hat_EKF_Prediction(2));
35
36      F = [1  , ...
37           0  , ...
38           dT , ...
39           0  ; ...
40
41           0  , ...
42           1  , ...
43           0  , ...
44           dT ; ...
45
46           0  , ...
47           -(dT*Ca*(l+r)/M)*x_hat_EKF_Prediction(4)*sin(x_hat_EKF_Prediction(2)) ,
                 ...
48           1-dT*Ca/M , ...
49           (dT*Ca*(l+r)/M)*cos(x_hat_EKF_Prediction(2)) ; ...
50
51           0  , ...
52           dT/J*( -M*g*(l+r)*cos(x_hat_EKF_Prediction(2)) - Ca*(l+r)* ...
53                  x_hat_EKF_Prediction(3)*sin(x_hat_EKF_Prediction(2)) +  ...
```

```
                Ca*((l+r)^2)*x_hat_EKF_Prediction(4)*
                cos(x_hat_EKF_Prediction(2))*sin(x_hat_EKF_Prediction(2)) + ...
                Fe*L*sin(x_hat_EKF_Prediction(2)))  , ...
        (dT/J*Ca*(l+r)*cos(x_hat_EKF_Prediction(2)))  , ...
        (1 - dT*Ca*((l+r)^2)/J*(cos(x_hat_EKF_Prediction(2))^2))];

    D = [0  0  dT/M -dT*L/J*cos(x_hat_EKF_Prediction(2))];

    P_EKF_Prediction = F*P_EKF_Prediction*F' + D*Q*D';
```

Per trovare le equazioni abbiamo portato il sistema in forma di stato e poi abbiamo discretizzato. Abbiamo raggruppato le 4 variabili di stato in un vettore $X$ in questo modo:

$$X = \begin{bmatrix} x_1 & x_2 & x_3 & x_4 \end{bmatrix} = \begin{bmatrix} x & \theta & \dot{x} & \dot{\theta} \end{bmatrix} \tag{12}$$

### 3.3.1 Regolarizzazione

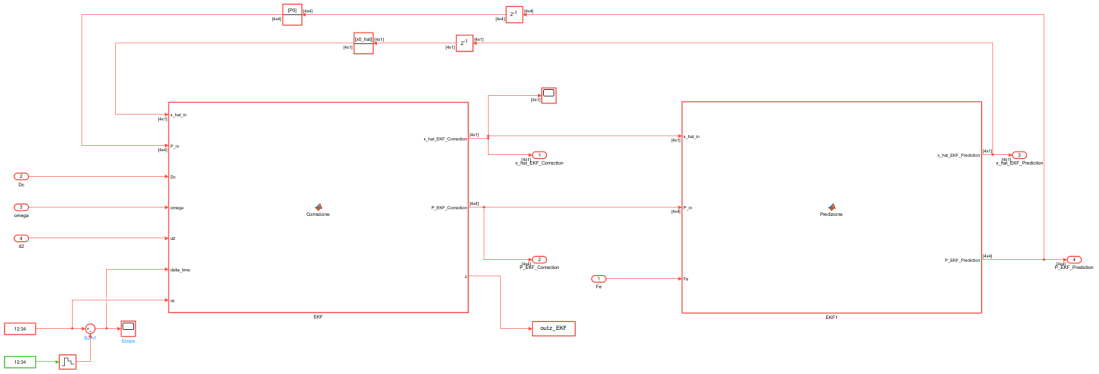
Per la regolarizzazione dell'EKF abbiamo applicato l'algoritmo di Rauch-Tung-Striebel (RTS) che abbiamo implementato in ambiente matlab con i dati provenienti dalla simulazione Simulink. Riportiamo sotto il codice di regolarizzazione:

Listing 7: Codice correzione PF

```
%% Regolarizzazione
Fe = funivia.Fe.Data;
%% Regolarizzazione EKF
x_hat_EKF_correction = funivia.x_hat_EKF_Correction.Data;
x_hat_EKF_prediction = funivia.x_hat_EKF_Prediction.Data;
P_EKF_correction = funivia.P_EKF_Correction.Data;
P_EKF_prediction = funivia.P_EKF_Prediction.Data;
F_EKF = funivia.F_EKF.Data;

[x_hat_EKF_smoothed, P_EKF_smoothed] = Smoothing (x_hat_EKF_correction,...
                                        x_hat_EKF_prediction,P_EKF_correction,...
                                        P_EKF_prediction,F_EKF);

%% Funzione di Regolarizzazione
function [x_hat_smoothed, P_smoothed] = Smoothing (x_hat_correction,...
                                            x_hat_prediction,P_correction,...
                                            P_prediction,F)
    x_hat_smoothed = x_hat_correction;
    P_smoothed = P_correction;
    Ck_plot = zeros (4,4,size(x_hat_correction,3)-1);
    %regolarizzo
    for k = size(x_hat_correction,3)-1:-1:1

        % Ck = P_k|k * F_k+1' * P_k+1|k^(-1)
        Ck = P_correction(:,:,k)*F(:,:,k)'/P_prediction(:,:,k);

        % x_k|n = x_k|k + Ck*(x_k+1|n - x_k+1|k)
        x_hat_smoothed(:,1,k) = x_hat_correction(:,1,k) +
            Ck*(x_hat_smoothed(:,1,k+1) - x_hat_prediction(:,1,k));

        % P_k|n = P_k|k + Ck*(P_k+1|n - P_k+1|k)*Ck'
        P_smoothed(:,:,k) = P_correction(:,:,k) +
            Ck*(P_smoothed(:,:,k+1)-P_prediction(:,:,k))*Ck';

    end

end
```

## 3.4 PF

Abbiamo implementato il Particle Filter in ambiente Simulink in questo modo:



Lo schema è uguale a quello dell'EKF, esclusa la retroazione che in questo caso contiene le particelle con i relativi pesi. Anche qui alleghiamo il codice della correzione:

Listing 8: Codice correzione PF

```
function [ x_hat_PF_correction, P_PF_correction, particles, weights, z] =
    Correzione (particles_in, weights_in, Dc, omega, d2, param, sensor, Fext,
    delta_time, ck)

M = param.M;
J = param.J;
Ca = param.Ca;
L = param.L;
l = param.l;
h = param.h;
g = param.g;
r = param.r;

particles = particles_in;
weights = weights_in;

likelyhood_for_particles = zeros(size(particles,1),1);

%% Correzione
if (delta_time <= 0) && (ck ~= 0)

    for i = 1:size(particles,1)

        likelyhood_Dc = normpdf(Dc - particles(i,1) +
            (l+r)*sin(particles(i,2)), 0 , sensor.Dc.std_dev);
        likelyhood_d2 = normpdf (d2 + (L-l-r)*sin(particles(i,2)), 0 ,
            sensor.d2.std_dev);
        likelyhood_omega = normpdf(omega - particles(i,3)/r +
            (l+r)/r*particles(i,4)*cos(particles(i,2)), 0 , sensor.Dc.std_dev);

        likelyhood_for_particles(i,1) =
            likelyhood_Dc*likelyhood_d2*likelyhood_omega;
    end

    weights = (weights.*likelyhood_for_particles)/...
              sum(weights.*likelyhood_for_particles);

    %ricampionamento
    s = cumsum(weights);
    epsilon = [0.1 0.01 0.02 0.02];
    new_particles = zeros(size(particles));

    for j = 1:size(particles,1)
        u = rand(1);
        csi = epsilon.*randn(1,4);
        i = find(s>=u,1,'first');
        new_particles(j,:) = particles(i,:) + csi;
    end
    new_weights = 1/size(weights,1) * ones(size(weights,1),1);

```

```
45          particles = new_particles;
46          weights = new_weights;
47          %
48
49          x_hat_PF_correction = particles'*weights;
50          P_PF_correction = zeros(4);
51          for i=1:size(particles,1)
52              P_PF_correction = P_PF_correction + weights(i)*(particles(i,1:4)' -
                    x_hat_PF_correction)*(particles(i,1:4)' - x_hat_PF_correction)';
53          end
54
55      else
56
57          x_hat_PF_correction = particles'*weights;
58          P_PF_correction = zeros(4);
59          for i=1:size(particles,1)
60              P_PF_correction = P_PF_correction + weights(i)*(particles(i,1:4)' -
                    x_hat_PF_correction)*(particles(i,1:4)' - x_hat_PF_correction)';
61          end
62
63      end
64
65      z = L*cos(x_hat_PF_correction(2));
```

Da notare che nel blocco di correzione si va a modificare solo i pesi e non le particelle. Abbiamo anche il ricampionamento, che in questo caso avviene ad ogni ciclo di correzione. Per la predizione il codice che abbiamo implementato è:

Listing 9: Codice predizione PF

```
1   function [particles, weights, x_hat_PF_prediction, P_PF_prediction] =
        Predizione (particles_in, weights_in, Fe, param, sensor, Fext, dT)
2
3       M = param.M;
4       J = param.J;
5       Ca = param.Ca;
6       L = param.L;
7       l = param.l;
8       h = param.h;
9       g = param.g;
10      r = param.r;
11
12
13      particles = zeros (size(particles_in));
14      weights = weights_in;
15      Q = Fext.std_dev^2;                                    % incertezza
            ingresso
16
17      R = [sensor.Dc.std_dev^2 0 0;...                       % matrice
            incertezze dei sensori
18           0 sensor.d2.std_dev^2 0;...
19           0 0 sensor.omega.std_dev^2];
20
21      %% Predizione
22      for i = 1:size(particles,1)
23          Wf = mvnrnd (0 , Q);
24          particles(i,1) = particles_in(i,1) + dT*particles_in(i,3);
25          particles(i,2) = particles_in(i,2) + dT*particles_in(i,4);
26          particles(i,3) = (1 - dT*Ca/M)*particles_in(i,3) +
                (dT*Ca*(l+r)/M)*particles_in(i,4)*cos(particles_in(i,2)) +
                (dT*(Fe+Wf)/M);
27          particles(i,4) = (1 -
                dT*Ca*((l+r)^2)/J*(cos(particles_in(i,2))^2))*particles_in(i,4) -
                (dT*M*g*(l+r)/J)*sin(particles_in(i,2)) +
                dT/J*(Ca*(l+r)*particles_in(i,3) - (Fe+Wf)*L)*cos(particles_in(i,2));
28
29      end
30
31       x_hat_PF_prediction = particles'*weights;
32       P_PF_prediction = zeros(4);
```

```
33      for i=1:size(particles,1)
34          P_PF_prediction = P_PF_prediction + weights(i)*(particles(i,1:4)' -
                x_hat_PF_prediction)*(particles(i,1:4)' - x_hat_PF_prediction)';
35      end
```

In questo caso notiamo che le equazioni[4] sono state applicate alle singole particelle e la predizione ha modificato solo queste ultime e non i pesi.

## 3.5 Applicazione su Modelli incerti

Per generare modelli incerti reali su cui applicare i filtri, abbiamo generato casualmente parametri che abbiamo limitato in dei range di variazione basati su dati di funivie reali.

### 3.5.1 Scelta parametri incerti

Per prima cosa abbiamo valutato quali parametri fosse ragionevole avere incerti nel nostro sistema.

**Lunghezze.** Possiamo supporre ragionevolmente che la costruzione della cabina e del "braccio" siano fatte in modo preciso, di conseguenza l'unica incertezza può essere data dalla dilatazione termica dell'alluminio[5], considerando che la temperatura ambientale possa variare tra $\pm 30°C$ vediamo che:

$$\delta l = l_{max} \Delta T \lambda = 0.00936m \tag{13}$$

dove $l_{max}$ è la lunghezza dal cavo di traino al fondo della cabina e $\lambda$ il coefficiente di dilatazione termica dell'alluminio. Questo valore è chiaramente trascurabile, quindi non abbiamo aggiunto incertezze.

**Massa e momento di inerzia.** La massa è ovviamente un parametro incerto, in quanto, supponendo una cabina in grado di caricare 20 persone, con una media di 70Kg a persona otteniamo una massa totale di 1400Kg che è più della metà del peso della cabina. Il momento di inerzia è dipendente dalla massa, quindi anche questo sarà incerto.

**Coefficiente di attrito.** Questo sarà dipendente, tra le altre cose, dalle condizioni meteo, per esempio il vento, quindi è ragionevole pensare che possa subire variazioni significative.

### 3.5.2 Inserimento incertezze

Per inserire le incertezze abbiamo sommato ai valori nominali una variabile aleatoria normale standard moltiplicata per un opportuno coefficiente, come mostrato nel codice:

```
1   % definizione Dati reali
2   var = randn*100;                                %variabile che usiamo per
3                                                   %far variare massa e
4                                                   %momento d'inerzia in modo
5                                                   %proporzionale, essendo
6                                                   %le due grandezze legate
7
8   param_real.M = param.M + var;                   %[kg]       %massa
9   param_real.J = param.J + var*(l+r)^2;           %[kg*m^2]   %momento di
10                                                              %inerzia
11  param_real.Ca = param.Ca + randn*3;             %[N*s/m]    %coefficiente
12                                                              %di attrito
13                                                              %viscoso
```

Come si può notare la massa ed il momento d'inerzia hanno a sommarsi la stessa variabile, essendo le due grandezze legate.

[4] ricavate come nell'EKF

[5] usato nella costruzione dei veicoli delle funivie secondo Leitner, noto costruttore di funivie

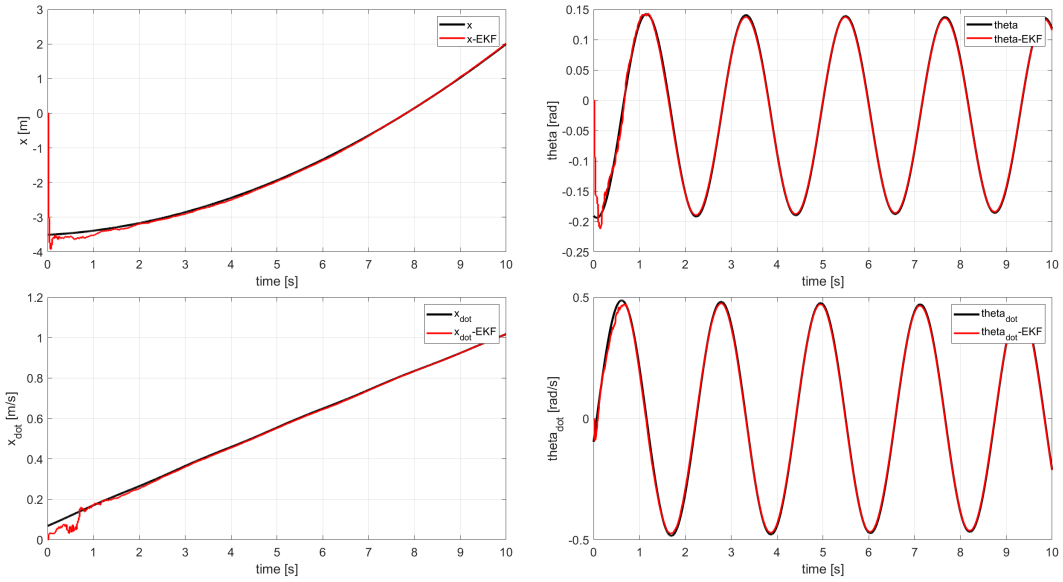



Figura 11: Stima dello stato EKF. $F = 200N$ costante

# 4 Test e Analisi risultati ottenuti

Per analizzare i risultati abbiamo utilizzato dati uscenti dai blocchi sia in ambiente Simulink sia su Matlab. Inoltre per visualizzare meglio i risultati abbiamo creato un'animazioone in grado di mostrare il movimento reale del sistema a confronto con quello dei filtri, questa sarà visibile su Matlab impostando su "true" la variabile "plot_dinamico". In particolare abbiamo eseguito i seguenti test sul sistema nominale:

1. **Ingresso costante.** Per questo test abbiamo impostato $F = 200N$ costante, in modo da raggiungere a regime la velocità che abbiamo ipotizzato, ovvero $10m/s$.

2. **Ingresso a gradino.** Successivamente, abbiamo impostato un ingresso a gradino di ampiezza $F = 500N$, in questo modo mettiamo alla prova le capacità di stima dei nostri filtri sia per una brusca variazione dell'ingresso, sia avendo come valore di regime un dato più "estremo". Inoltre sarà visibile una fase iniziale dove il sistema, partendo inclinato, oscillerà libero.

3. **Ingresso a rampa.** Infine abbiamo provato a impostare un ingresso a rampa, abbiamo pensato che una pendenza di $30N/s$ fosse sufficiente.

Per ogni test abbiamo impostato un tempo di simulazione che ritenevamo opportuno per riuscire ad apprezzare il transitorio del filtro e successivamente il suo comportamento quando raggiunge lo stato del sistema.

## 4.1 EKF

**Test F = 200N.** Nella figura 11 possiamo osservare che, dopo un breve transitorio iniziale di durata accettabile considerando il tipo di sistema, il filtro si assesta sullo stato del sistema commettendo un errore accettabile rispetto ai sensori utilizzati ed al tipo di sistema.

**Test F a gradino.** Nella figura 12 possiamo osservare, anche in questo caso, che abbiamo un breve transitorio iniziale, dopo il quale il filtro compie un'ottima stima. Interessante notare che l'arrivo del gradino non influenzi l'errore commesso dal filtro.

**Test F a rampa.** Nella figure 13 e 14 la stima converge allo stato reale in modo piuttosto rapido anche in questo caso.

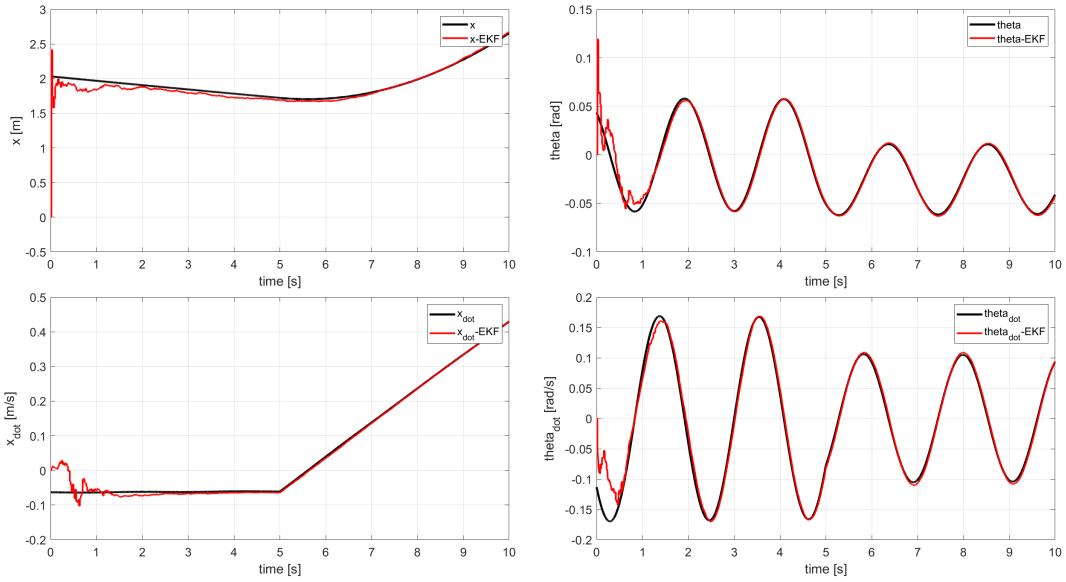



Figura 12: Stima dello stato EKF. ingresso a gradino

### 4.1.1 Regolarizzazione

In questo capitolo analizziamo il funzionamento dell'algoritmo di regolarizzazione per l'EKF. Chiaramente, vedendo i risultati ottenuti nei test precedenti ci aspettiamo che la stima regolarizzata si discosti da quella senza regolarizzazione nei punti dove questa ha dei gradini più ampi dovuti alle correzioni, in particolare le correzioni più grandi le abbiamo ad inizio simulazione visto che il filtro, che parte da uno stato casuale, dovrà avvicinarsi allo stato del sistema. Nelle immagini 15, 16 e 17vediamo i risultati, che coincidono con le nostre aspettative.

## 4.2 PF

Anche per il Particle Filter abbiamo ripetuto gli stessi test, i risultati sono visibili nelle figure 18, 19 e 20:

**Test F = 200N.** Anche in questo caso gli stati stimati convergono rapidamente a quelli del sistema, questo coincide con le nostre aspettative.

**Test F a gradino.**

**Test F a rampa.** Notiamo anche in questo caso che dopo l'incertezza iniziale, la stima converge in un intorno dello stato reale compatibile con la precisione dei sensori.

## 4.3 Confronto Filtri

Mettiamo ora a confronto i due filtri notando differenze e similitudini sui vari aspetti. Abbiamo messo a confronto la stima di tutte le variabili di stato. Inizialmente abbiamo impostato il PF con 5000 particelle e dai risultati ottenuti (vedi figura 21) possiamo apprezzare che la due stime convergono entrambe alla stima vera ma con diverse velocità. Infatti il PF impiega più tempo a convergere. Questo può esser dovuto al fatto che la linearizzazione del sistema non fa commettere un grosso errore, il che permette all'EKF di funzionare al meglio. Inoltre tutti i rumori che abbiamo inserito sono gaussiani e l'EKF è progettato per lavorare proprio con questo tipo di distribuzione. Allo stesso tempo però per tutti gli stati rimaniamo all'interno dei range di possibile errore di misura dei sensori. Quindi in questo caso è preferibile l'EKF in quanto oltre ad un errore minore, è più leggero a livello computazionale. A riprova di quest'ultima osservazione, abbiamo inserito

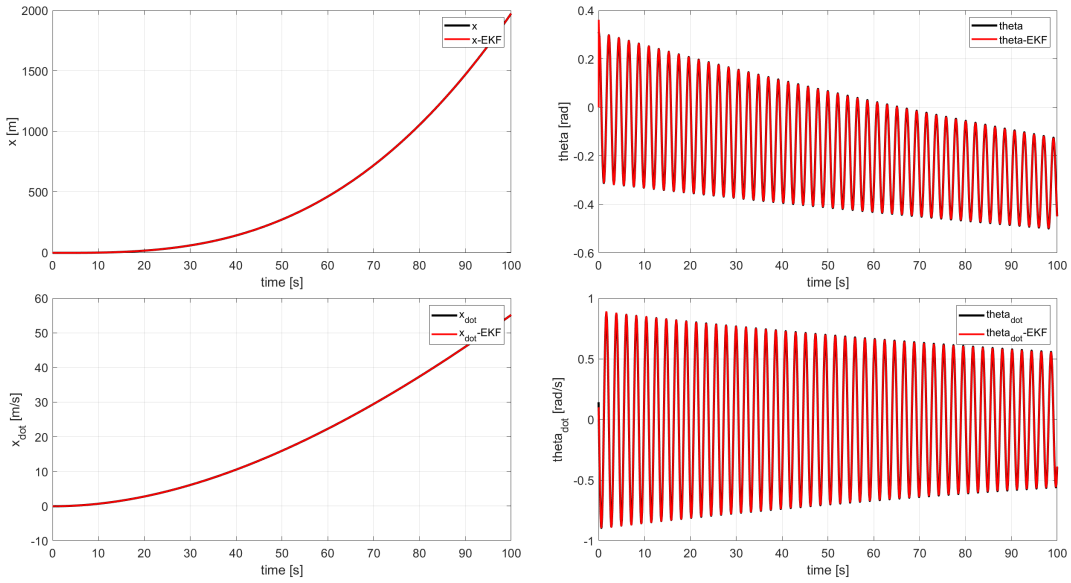



Figura 13: Stima dello stato EKF. ingresso a rampa, tempo di simulazione 100s

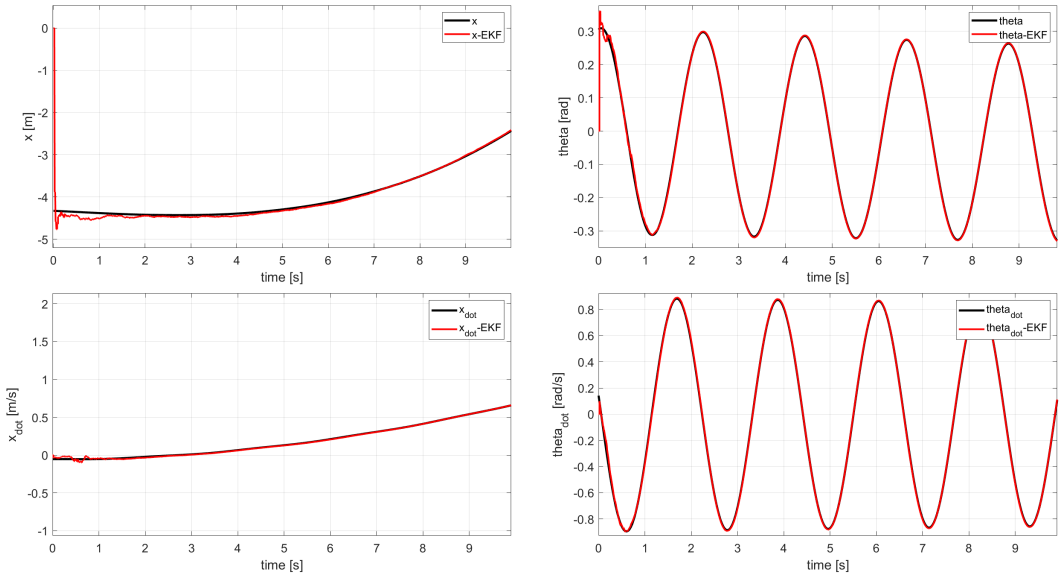



Figura 14: Stima dello stato EKF. ingresso a rampa, tempo di simulazione 10s

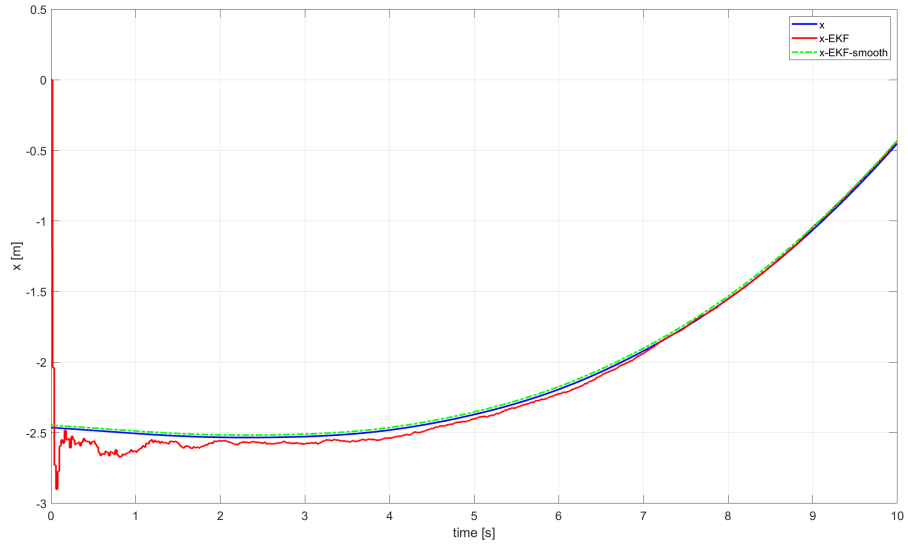



Figura 15: posizione della funivia

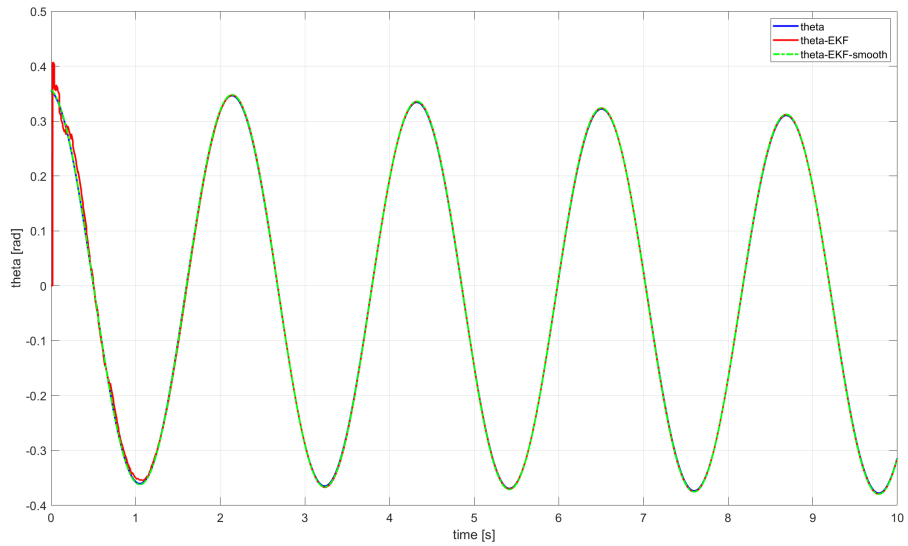



Figura 16: inclinazione della funivia

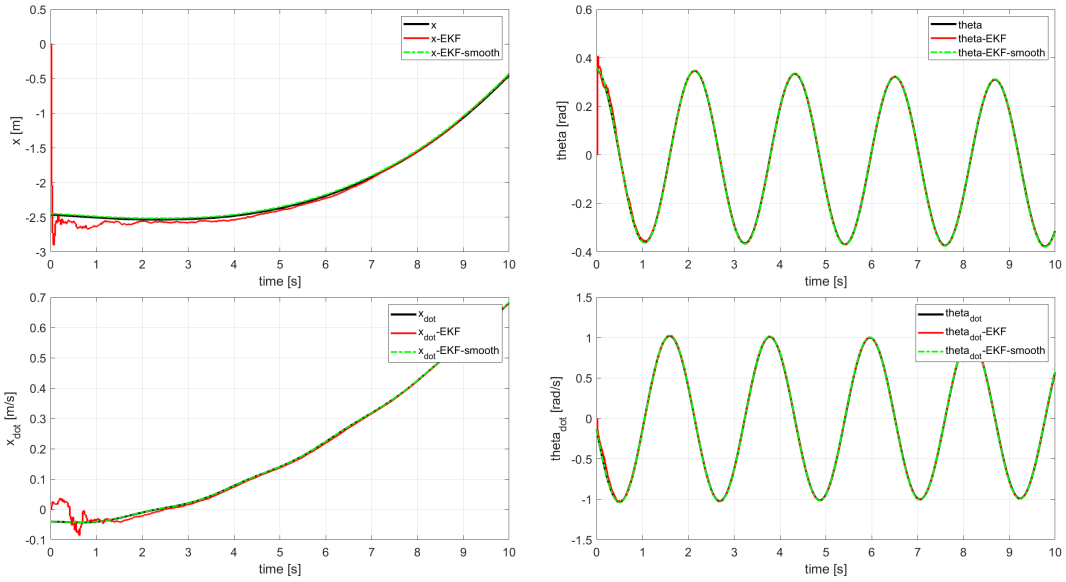



Figura 17: tutte le variabili di stato

anche due simulazioni variando il numero di particelle del Particle Filter portandole rispettivamente a 1000 (figura 22) e 10000 (figura 23). Da queste prove possiamo notare come la stima migliori all'aumentare delle particelle, ma con questo anche lo sforzo computazionale. Infatti, abbiamo registrato un tempo di simulazione totale[6] di circa 14 secondi impostando il numero di particelle a 5000, di circa 40 secondi con 10000 particelle e di circa 5 secondi con 1000 particelle. Avendo un piccolo miglioramento di stima in paragone al grande aumento computazionale, è preferibile la scelta di un numero minore di particelle, dato che la stima rimane comunque all'interno dei range dati dai limiti dei sensori di misura.

[6]la simulazione include il modello, l'EKF ed il PF

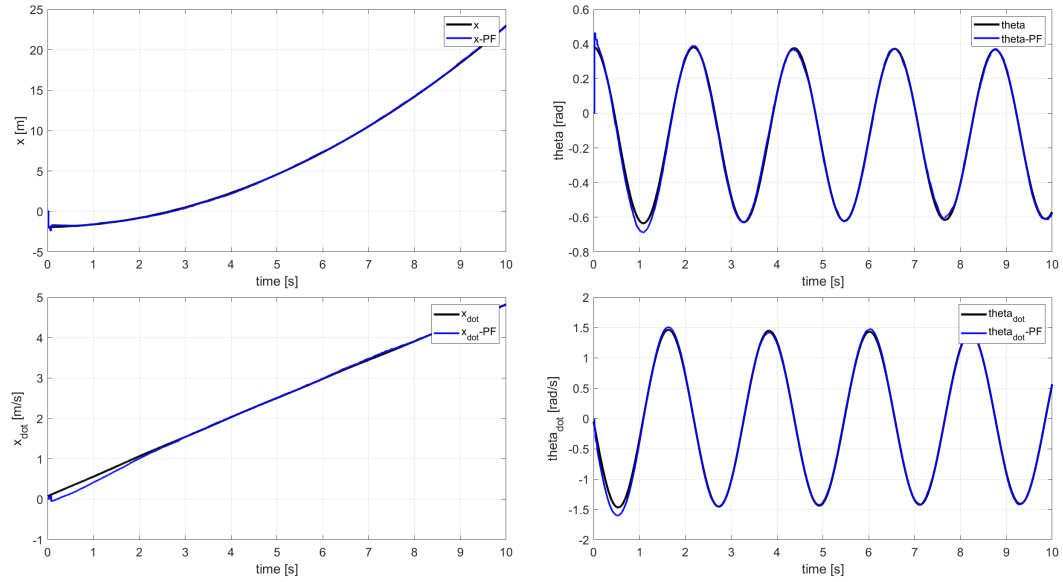



Figura 18: Stima dello stato PF. $F = 200N$ costante

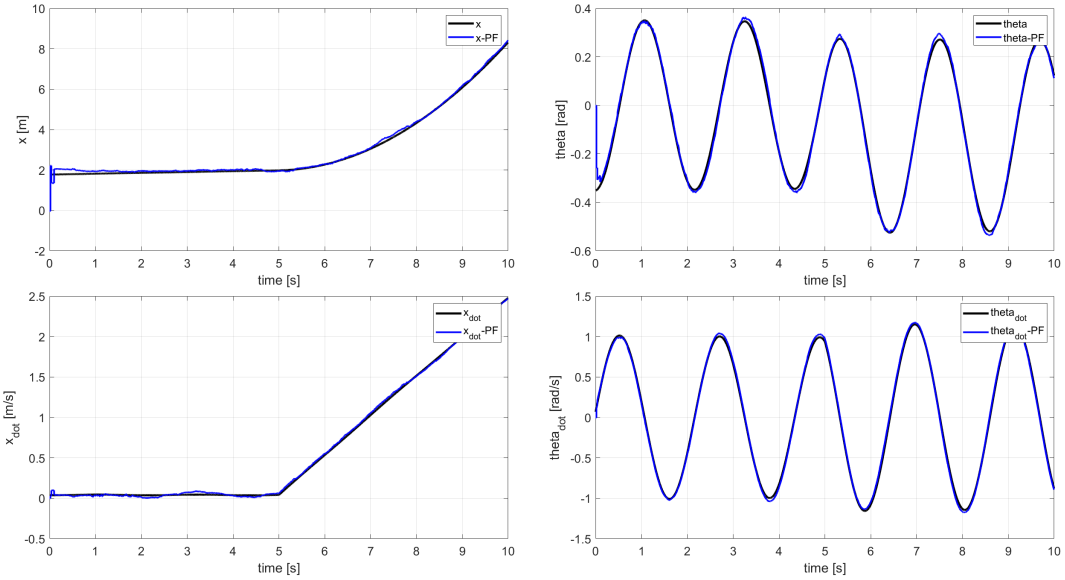



Figura 19: Stima dello stato PF. ingresso a gradino

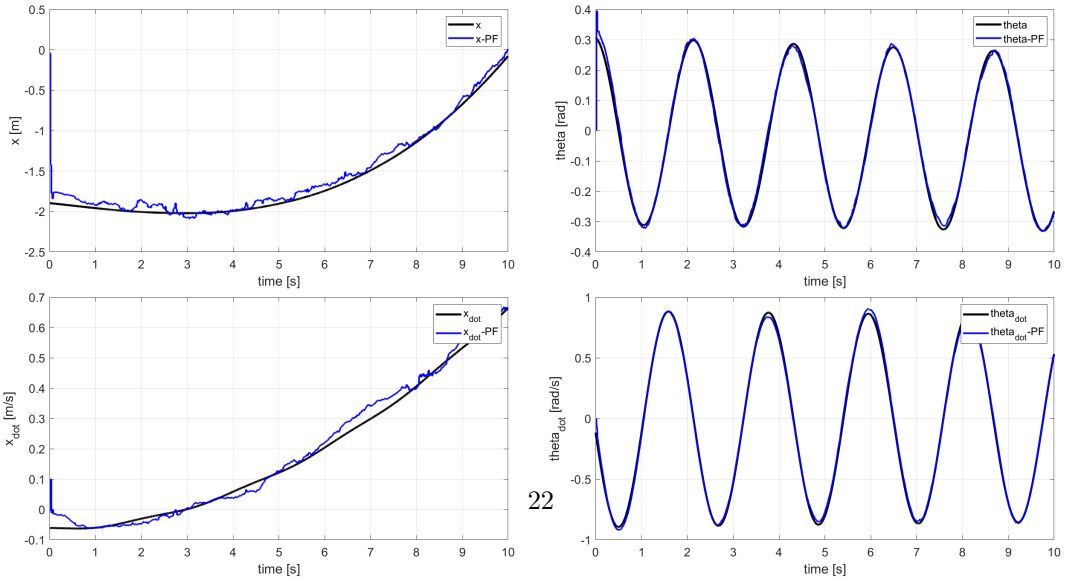





Figura 20: Stima dello stato PF. ingresso a rampa

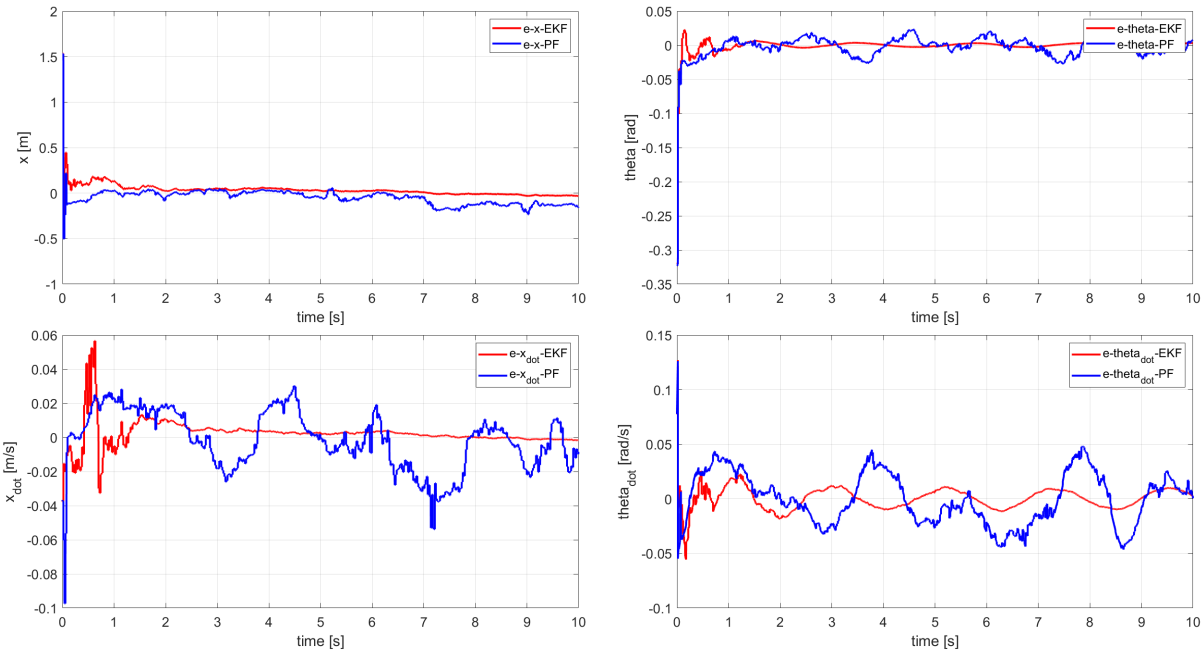



Figura 21: errore commesso dall'EKF e dal PF (5000 particelle)

## 4.4 Modello incerto

Per analizzare il funzionamento in un contesto più realistico, abbiamo simulato il modello con i parametri incerti con $F = 200N$ costante, i due filtri continuano a stimare gli stati con un'errore limitato, tuttavia come possiamo notare dalle immagini 24 e 25 la stima del PF risulta nettamente migliore. Una possibile spiegazione sta nel fatto che in generale il particle filter è più adatto a gestire non linearità o dinamiche complesse del sistema in esame rispetto all'EKF. Inoltre la matrice di covarianza P tende a diventare molto piccola nelle iterazioni, quindi le correzioni influenzeranno meno la stima e la differenza tra i parametri nominali e reali tenderà a far aumentare l'errore.

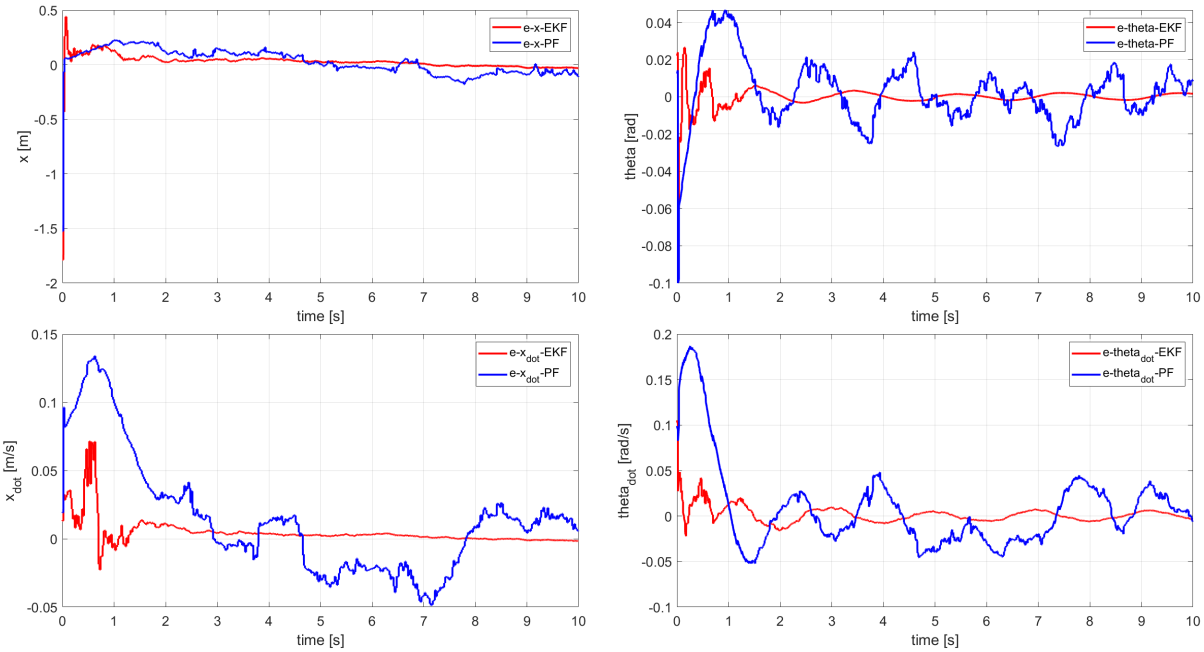



Figura 22: errore commesso dall'EKF e dal PF (1000 particelle)

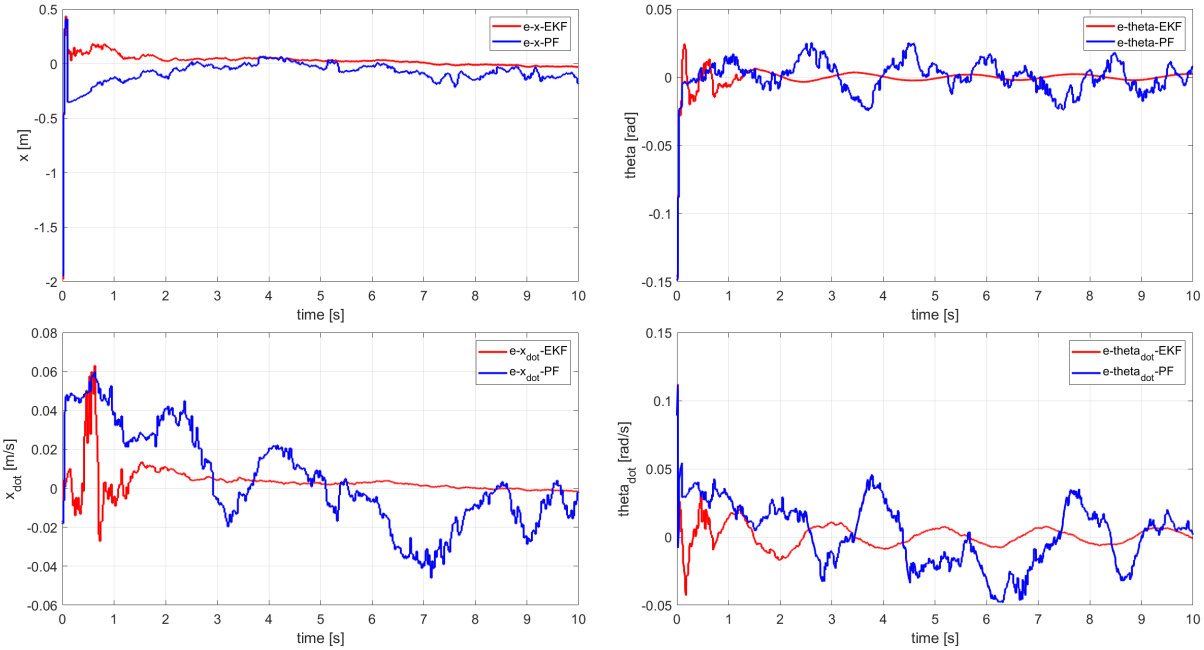



Figura 23: errore commesso dall'EKF e dal PF (10000 particelle)

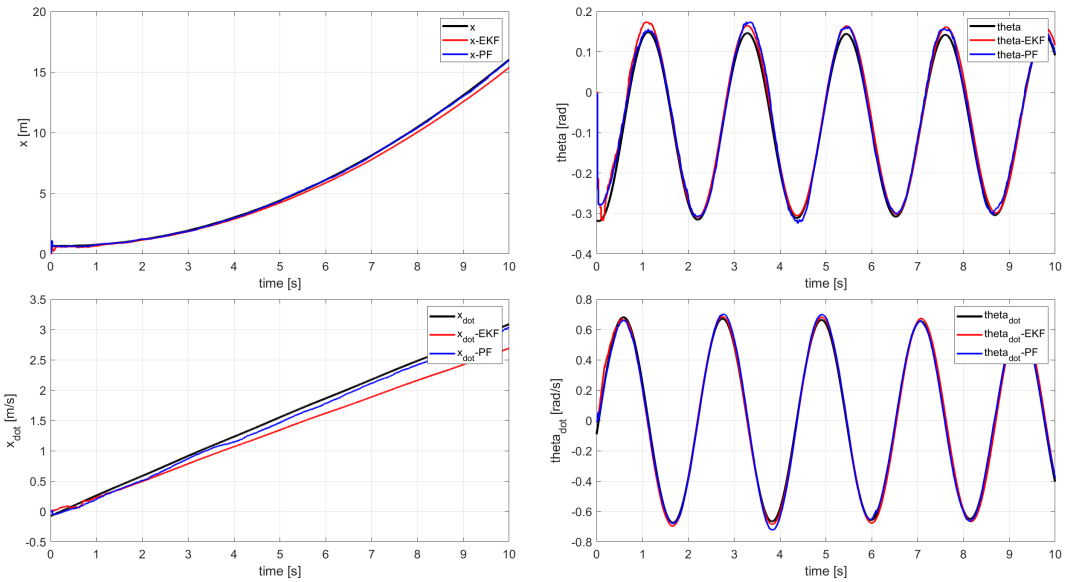



Figura 24: stima delle variabili di stato EKF e PF. ingresso F = 200N costante, tempo di simulazione 10s, modello reale

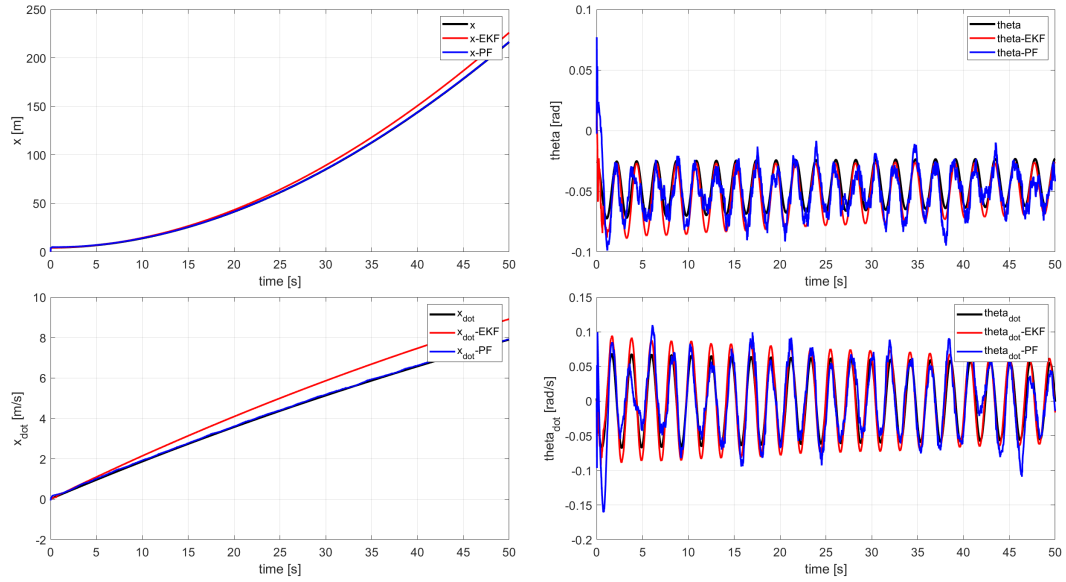



Figura 25: stima delle variabili di stato EKF e PF. ingresso F = 200N costante, tempo di simulazione 50s, modello reale