Università degli Studi di Padova

Facoltà di Ingegneria
Dipartimento di Ingegneria dell'Informazione
Corso di Laurea in Ingegneria dell'Informazione

Tesi di Laurea in Ingegneria dell'Informazione

Co-simulazione di un Sistema di Controllo

Relatore: Prof. Fabio Marcuzzi

Laureando: Paolo Martin

A.A. 2011-2012

Abstract

La *co-simulazione* si occupa della simulazione collaborativa del modello del sistema fisico e del firmware del microcontrollore, modellando le interazioni tra i vari componenti.
In questa tesi si punta ad approfondire i vari aspetti di una co-simulazione, con l'obiettivo di arrivare a simulare un sistema di controllo nella sua completezza.
Dopo un'analisi iniziale su costi e benefici della co-simulazione, si introdurranno le equazioni differenziali ordinarie, grazie alle quali è possibile rappresentare matematicamente molti sistemi fisici. Si focalizzerà l'attenzione sull'analisi di stabilità e sul confronto dei vari metodi di risoluzione, con particolare riguardo per i problemi *stiff*.
Successivamente si prenderà in considerazione il sistema di controllo, che sarà formato da un microcontrollore. Verrà presentata una possibile discretizzazione del controllore PID e si discuterà l'algoritmo di taratura *Trial-and-Error*.
Seguirà infine la co-simulazione all'interno del software μ*Lab* di un esempio di sistema meccanico controllato da un microcontrollore PIC. Le equazioni che modellano il sistema meccanico verranno risolte utilizzando le librerie *SUNDIALS* tramite il wrapper *PySUNDIALS*.

Indice

1 La Co-Simulazione **1**

1.1 Benefici . . . 3

1.2 Costi . . . 3

1.3 Il software μLab® . . . 4

2 Equazioni Differenziali Ordinarie **5**

2.1 Il Problema di Cauchy . . . 5

2.2 Controllo degli errori . . . 7

2.3 Stabilità e convergenza . . . 7

2.4 Metodi ad un passo . . . 10

2.5 Metodi multistep . . . 12

2.6 Metodi BDF . . . 15

2.7 Problemi stiff . . . 17

2.8 Risoluzione di sistemi algebrici non lineari . . . 21

3 Descrizione del Sistema **23**

3.1 Il sistema fisico . . . 24

3.2 L'attuatore . . . 26

3.3 L'amplificatore differenziale . . . 26

3.4 Il generatore di traiettorie . . . 27

3.5 Il controllore . . . 27

4 Il Controllo **29**

4.1 Il controllo PID . . . 30

4.2 Discretizzazione e algoritmo di posizione . . . 32

4.3 Risultati delle simulazioni . . . 33

5 Conclusioni **39**

A Codice sorgente **41**

A.1 Modello Python 3 masse . . . 41

A.2 Modello Python attuatore 43
A.3 Modello Python generatore di traiettorie 43
A.4 Modello Python amplificatore differenziale 44
A.5 Firmware del microcontrollore in C 44
A.6 Codice Python per esempio problemi stiff 46

Bibliografia **51**

Capitolo 1

La Co-Simulazione

La progettazione di sistemi complessi richiede una verifica del loro corretto funzionamento, verifica che può essere portata a termine principalmente in due modi, cfr. [11], cap. 11.

La verifica formale è un'analisi volta a provare analiticamente la correttezza dell'implementazione del sistema di controllo. Ad esempio è possibile dimostrare l'incompatibilità di due eventi interni al sistema. Questa verifica è la più precisa e completa ma richiede tempi molto lunghi ed è perciò applicabile solo a piccole parti del sistema. Inoltre, non è possibile valutare in tutta generalità le interazioni del sistema di controllo con l'ambiente multifisico in cui opera, e quindi le sue prestazioni funzionali.

L'alternativa a questo tipo di verifica è la *simulazione*. Questo approccio è basato sulla simulazione al calcolatore del sistema in questione, che può essere anche molto complesso.

Nel caso il sistema in questione possieda uno o più microcontrollori parliamo di *co-simulazione*, intendendo la simulazione complessiva del modello numerico multifisico e del firmware in esecuzione, con particolare attenzione alle loro interazioni.

In una co-simulazione quindi entrano in gioco la simulazione dei modelli che rappresentano i vari componenti del sistema, la simulazione dell'ambiente esterno (anch'essa attraverso modelli) e la simulazione dell'esecuzione del firmware, necessariamente effettuata all'interno di un modello del microcontrollore.

La modellizzazione del microcontrollore è particolarmente importante, deve essere un compromesso in cui viene garantita la correttezza logica del funzionamento ma non quella fisica, dal momento che risulta inutile simulare i fenomeni fisici ad un livello di dettaglio poco rilevante per la valutazione del comportamento del sistema, soprattutto se è necessario un tempo lun-

ghissimo (secondo una stima in [11] per simulare per un'ora una piattaforma embedded a livello di porta logica ci vogliono più di mille anni).

Inoltre si è interessati al funzionamento interno di un dispositivo solo in fase di progettazione. Si prenda ad esempio una rete che realizza una funzione logica. Durante la sua progettazione è necessario approfondire nel dettaglio tutti gli aspetti delle singole porte logiche, dopo averla realizzata e dopo averne verificato il funzionamento invece tutto ciò può essere tralasciato dato che ciò che è effettivamente importante è il valore dell'uscita in corrispondenza di una particolare configurazione di ingressi.

Passando ai vari modelli, la co-simulazione prevede che ogni componente venga risolto da un apposito simulatore, che lavora in maniera indipendente dagli altri tranne quando è necessario uno scambio di informazioni tra i componenti. Ad esempio, un modello scritto in linguaggio Python verrà risolto dall'apposito interprete Python, che, quando necessario, resterà in attesa dei dati in ingresso dagli altri componenti.

I componenti interagiscono tra di loro tramite un piccolo insieme di variabili fisiche, un semplice esempio può essere l'azionamento di un motore elettrico. Si possono distinguere due blocchi, l'attuatore e il motore vero e proprio, il motore deve ricevere dall'attuatore il valore di corrente che circolerà in esso.

Per gestire al meglio le interazioni tra componenti diversi è buona norma definire un passo di simulazione comune, dopo il quale ogni componente aggiorna il valore delle proprie variabili di uscita. A questo proposito si deve prima definire il livello di dettaglio del modello della piattaforma embedded. Normalmente non si è interessati a cosa succede a livello delle singole porte logiche e si ritiene sufficiente una simulazione basata sul set di istruzioni del microprocessore e sugli effetti della loro esecuzione. Un primo vincolo inferiore al passo di simulazione è quindi dato dal tempo di esecuzione dell'istruzione più breve.

Gli altri componenti solitamente sono discretizzazioni di fenomeni continui nel tempo. Questi fenomeni sono descritti da sistemi di equazioni differenziali (o algebrico-differenziali) che possono essere risolte numericamente con i metodi descritti nel capitolo 2. Esistono delle librerie che implementano già alcuni di questi metodi, tra le quali si citano le SUNDIALS (SUite of Nonlinear and DIfferential/ALgebraic equation Solvers, `https://computation.llnl.gov/casc/sundials/main.html`), che contengono il pacchetto CVODE [6] con i metodi per la risoluzione di sistemi di equazioni differenziali stiff e non-stiff (cfr. 2.7). Il pacchetto comprende anche metodi per la risoluzione del sistema algebrico non-lineare necessario nella soluzione dei problemi stiff (cfr. 2.8).

1.1 Benefici

Lavorare in un ambiente co-simulato presenta vari vantaggi, uno su tutti la possibiltà di conoscere quantitativamente il comportamento del sistema in risposta ad un determinato evento, dovuto ad una particolare condizione fisica del sistema o all'esecuzione di un'istruzione del firmware.

Tra i vantaggi maggiori ci sono i seguenti:

- La possibilità di controllare l'esecuzione ad un livello molto profondo. È possibile avviare e fermare la simulazione in qualsiasi momento, modificare parametri del sistema e dell'ambiente esterno, valori di variabili e soprattutto è possibile avanzare la simulazione a passi molto ridotti, ad esempio dell'ordine dei microsecondi. Risulta evidente l'impossibilità fisica di raggiungere tali obiettivi su un sistema reale.

- La possibilità di esaminare in ogni istante i valori caratteristici del sistema. Una simulazione non rende disponibili solo determinati parametri come può fare un sistema fisico, permette di visualizzare anche i valori dei registri del microcontrollore e delle variabili di stato dei vari modelli.
  Quest'ultima possibilità risulta fondamentale quando si svolge un'attività di *debugging*.

Altri vantaggi di una co-simulazione sono dati in termini temporali, il tempo necessario per creare la simulazione solitamente è molto inferiore al tempo che si impiega per la realizzazione fisica del sistema.

Una simulazione del sistema è anche più sicura, soprattutto durante le prime fasi della progettazione. Se il sistema fisico non dovesse funzionare correttamente al primo tentativo c'è la possibilità che si danneggi, eventualità assente (ovviamente) in una simulazione.

1.2 Costi

La co-simulazione include anche il costo aggiuntivo di rendere necessaria la costruzione dei modelli dei vari componenti del sistema.

Va puntualizzato che il modello, per quanto possa somigliare alla realtà, sarà sempre approssimato e quindi l'errore di approssimazione rappresenta un fenomeno in più da tenere conto e controllare.

Un'altra questione fondamentale è la velocità di simulazione. Simulare un sistema può richiedere molto più tempo di un'esecuzione reale, inoltre non è

possibile simulare il comportamento del sistema per ogni possibile sequenza di input, richiederebbe un tempo troppo lungo. È fondamentale identificare un insieme di casi da simulare, che solitamente comprende alcuni esempi ed i casi limite. Significa dunque avere un grado di copertura potenzialmente inferiore rispetto all'analisi formale. L'efficacia dipende dunque anche dalla capacità di progettare test significativi ed esaustivi, come avviene nella verifica sperimentale dei sistemi.

1.3 Il software μLab®

Per la co-simulazione verrà utilizzato il software *μLab*, che permette di simulare una piattaforma embedded contenente fino a due microcontrollori interconnessi tra di loro e con dei componenti esterni scritti in linguaggio Python. *μLab* simula il funzionamento dell'intero sistema eseguendo le istruzioni contenute nel file binario prodotto dal compilatore del firmware per il microprocessore e fornisce una vista globale sulle variabili in gioco e sui registri dei microcontrollori. Permette inoltre la sostituzione di intere parti del firmware con subroutines in Python, possibilità molto utile nelle prime fasi di progettazione del sistema.

Capitolo 2

Equazioni Differenziali Ordinarie

Le equazioni differenziali ordinarie, dette anche ODE (Ordinary Differential Equations) sono delle equazioni che coinvolgono una variabile indipendente t, una variabile dipendente y ed almeno una derivata, non necessariamente del primo ordine, di y rispetto a t. L'ordine di un'equazione differenziale è l'ordine massimo delle derivate che vi compaiono.

Un'equazione differenziale ordinaria di ordine p è esprimibile come

$$y^{(p)}(t) = \Phi\left(t, y(t), y'(t), ..., y^{(p-1)}(t)\right) \tag{2.1}$$

e può essere trasformata in un sistema di p equazioni di ordine uno tramite la definizione di variabili ausiliarie che rappresentano le derivate della variabile indipendente ponendo

$$z_1(t) = y(t), \quad z_2(t) = y'(t), \quad ... \quad z_p(t) = y^{(p-1)}(t) \tag{2.2}$$

Il problema (2.1) diventa

$$\begin{cases} z_1'(t) = z_2(t) \\ ... \\ z_{p-1}'(t) = z_p(t) \\ z_p'(t) = \Phi\left(t, z_1(t), ..., z_{p-1}(t)\right) \end{cases} \tag{2.3}$$

Risolvere un'equazione di ordine p equivale quindi a risolvere un sistema di p equazioni di ordine uno.

2.1 Il Problema di Cauchy

In generale un'equazione differenziale ordinaria di ordine uno ammette infinite soluzioni. Per identificare univocamente la soluzione del problema è

necessario imporre il valore assunto in un punto dell'intervallo di integrazione, usualmente il punto iniziale. Tale valore viene detto in tal caso *condizione iniziale.*

Risolvere un problema di Cauchy significa trovare l'unica soluzione $y : I \to \mathbb{R}$ di

$$\begin{cases} y'(t) = f(t, y(t)) & \forall t \in I \\ y(t_0) = y_0 \end{cases} \tag{2.4}$$

dove I è un intervallo di $\mathbb{R}$, $f : I \times \mathbb{R} \to \mathbb{R}$ una funzione assegnata e y' indica la derivata di y rispetto a t. Infine, t_0 è un punto di I e y_0 è la condizione iniziale.

Una funzione $f(t, y)$ continua su $I \times \mathbb{R}$ soddisfa la *condizione di Lipschitz* nella variabile y se esiste una costante reale positiva L tale che:

$$\|f(t, y_1) - f(t, y_2)\| \leq L\|y_1 - y_2\|, \qquad \forall (t, y_1), (t, y_2) \in I \times \mathbb{R} \tag{2.5}$$

Si dimostra (cfr. [10] e [1]) che se un problema di Cauchy ha la funzione *lipschitziana* rispetto al secondo argomento, allora la soluzione esiste, è unica e di classe C^1 su I.

La risoluzione di un'equazione differenziale ordinaria (di ordine uno) passa attraverso l'integrazione della funzione $f(t, y)$:

$$y(t) - y(t_0) = \int_{t_0}^{t} y'(\tau)\, d\tau = \int_{t_0}^{t} f(\tau, y)\, d\tau \tag{2.6}$$

Per alcune equazioni differenziali è possibile calcolare in forma esplicita la soluzione dell'integrale, per altre invece ci si deve accontentare della forma implicita, per altre ancora si rende necessario uno sviluppo in serie.

Per poter risolvere qualsiasi tipo di equazione differenziale è necessario utilizzare un metodo numerico che approssimi l'andamento della soluzione reale $y(t)$.

Per applicare un metodo numerico è necessario dividere l'intervallo di integrazione $[t_0, T]$, con $t < +\infty$ in N_h intervalli. Si definisce *passo di discretizzazione* $h = (T - t_0)/N_h$.

La soluzione numerica sarà formata da una successione di valori u_n che approssimano la funzione $y(t_n)$ ai nodi $t_n = t_0 + nh$ (con $n = 0, 1, 2...$).

2.2 Controllo degli errori

Il fatto che il metodo numerico approssimi l'andamento della soluzione esatta lascia intendere che esso commetterà un certo errore.
Gli errori commessi durante la risoluzione del problema ai valori iniziali possono essere divisi in due categorie:

- errori di arrotondamento
- errori di approssimazione per troncamento

Gli errori di arrotondamento sono attribuibili al fatto che i calcolatori hanno la necessità di discretizzare i numeri reali per poterli rappresentare. Ci sono quindi necessariamente alcuni numeri che non possono essere rappresentati e vengono approssimati al valore più vicino.

Per quanto piccoli si vedrà che questi errori possono assumere un ruolo chiave nella risoluzione di alcune tipologie di problemi.

Gli errori di approssimazione invece sono errori prodotti dal metodo numerico stesso e, come dice il nome, sono dovuti ai troncamenti necessari nello sviluppo della funzione in serie di Taylor e nelle interpolazioni polinomiali.

Solitamente l'*errore di troncamento locale* (LTE) viene definito come

$$\tau_n(h) = \frac{(y(t_n) - u(t_n))}{h} \tag{2.7}$$

dove $y(t_n)$ è la soluzione esatta, $u(t_n)$ la soluzione calcolata al passo n partendo dalla soluzione esatta a t_{n-1} e h è il passo di discretizzazione.

Si definisce anche l'*errore di troncamento globale* $\tau(h)$ come l'LTE massimo nell'intervallo $t_0, ..., t_{N_h-1}$

2.3 Stabilità e convergenza

Ciò che si vuole da un metodo numerico è un risultato quanto più possibile aderente alla realtà. Si rende necessario quindi uno studio per evidenziare i parametri che consentono di individuare un buon metodo numerico.

La proprietà fondamentale di un metodo numerico è la convergenza. Un metodo è convergente se per ogni problema ai valori iniziali con funzione lipschitziana e per ogni $t \in [t_0, T]$ si ha

$$\lim_{h \to 0} u(t_n) = y(t_n) \tag{2.8}$$

In altri termini, la soluzione calcolata deve tendere, per h che tende a zero, alla soluzione del problema continuo. Una formulazione alternativa porta a

$$\lim_{h\to 0} \max_{0\leq n\leq N_h} |y(t_n) - u(t_n)| = 0 \tag{2.9}$$

La convergenza è assolutamente necessaria in un metodo numerico, altrimenti la soluzione perde qualsiasi significato. Ora è necessario stabilire quali sono le condizioni che portano alla convergenza del metodo.

Un buon punto di partenza è l'errore di approssimazione $y(t_n)-u(t_n)$. Se l'approssimazione $u(t_n)$ coincide con la soluzione teorica $y(t_n)$ si ottiene un errore nullo. Questo suggerisce l'interpretazione di tale errore come misura della qualità della soluzione.

Richiedere solo la convergenza a zero di $y(t_n) - u(t_n)$ però non appare sufficiente, si cerca un legame con il passo di discretizzazione h. Si arriva quindi alla proprietà di consistenza:

Un metodo numerico si dice *consistente* con il problema di Cauchy quando il suo errore di troncamento locale è infinitesimo rispetto ad h, in termini matematici

$$\lim_{h\to 0} \tau_n(h) = 0 \tag{2.10}$$

È importante considerare anche la stabilità di un metodo numerico, intesa come possibiltà di controllare l'effetto sulla soluzione di una perturbazione dei dati. Si vuole infatti che a piccole perturbazioni del problema il metodo produca perturbazioni limitate sull'uscita. Questa proprietà permette di tenere sotto controllo gli errori di arrotondamento introdotti dall'aritmetica finita dei calcolatori.

Consideriamo un problema del tipo

$$\sum_{m=0}^{s} a_m u_{n+m} - h \sum_{m=0}^{s} b_m f(t_{n+m}, u_{n+m}) = 0, \qquad n = 0, 1, ... \tag{2.11}$$

Verificare la proprietà di zero-stabilità corrisponde a studiare il comportamento del metodo per $h = 0$, in particolare si studia l'effetto locale di una perturbazione del problema.

Da [9], un metodo ad un passo si dice *zero-stabile* se

$$\begin{aligned} &\exists h_0 > 0, \exists C > 0, \exists \epsilon_0 > 0 \text{ tali che } \forall h \in (0, h_0] \text{ e } \forall \epsilon \in (0, \epsilon_0], \\ &\text{se } |\delta_n| \leq \epsilon, 0 \leq n \leq N_h, \text{ allora } \quad |z_n^{(h)} - u_n^{(h)}| \leq C\epsilon, \quad 0 \leq n \leq N_h \end{aligned} \tag{2.12}$$

dove $z_n^{(h)}$ e $u_n^{(h)}$ sono rispettivamente le soluzioni dei problemi

$$\begin{cases} z_{n+1}^{(h)} = z_n^{(h)} + h\left[\Phi\left(t_n, z_n^{(h)}, f\left(t_n, z_n^{(h)}; h\right)\right) + \delta_{n+1}\right], & n = 0, ..., N_h - 1 \\ z_0^{(h)} = y_0 + \delta_0 \end{cases} \tag{2.13}$$

$$\begin{cases} u_{n+1}^{(h)} = u_n^{(h)} + h\left[\Phi\left(t_n, u_n^{(h)}, f\left(t_n, u_n^{(h)}; h\right)\right)\right], & n = 0, ..., N_h - 1 \\ u_0^{(h)} = y_0 \end{cases} \tag{2.14}$$

Esiste un metodo più immediato che consente di verificare la zero-stabilità, si tratta della *condizione delle radici.*
Un metodo la verifica se tutte le radici w_i del suo polinomio caratteristico $\rho(w) = \sum_{j=0}^{k} a_j w^j$ sono dentro al cerchio unitario e quelle sulla circonferenza sono semplici.

Il teorema di equivalenza di Lax-Richtmyer assicura che condizione necessaria e sufficiente per la convergenza di un metodo numerico zero-stabile è che esso sia consistente [3].

Si definisce inoltre *ordine di convergenza* del metodo il valore $p > 0$ per il quale $|y(t_n) - u(t_n)| \leq C(h)$ per ogni n, con $C(h) = \mathcal{O}(h^p)$.

Inoltre, in molte applicazioni si è interessati alla risolvere il problema ai valori iniziali per tempi molto grandi mantenendo comunque una certa accuratezza. Il metodo numerico quindi deve garantire una soluzione *buona* anche quando si utilizza un passo h ampio.

Non tutti i metodi possono garantire questo, si consideri ad esempio il problema

$$\begin{cases} y'(t) = \lambda y(t), & t \in (0, \infty) \\ y(0) = 1 \end{cases} \tag{2.15}$$

La soluzione teorica del problema è $y(t) = e^{\lambda t}$, che converge a zero quando $t \to \infty$. Si ha dunque la convergenza se e solo se

$$\lim_{n \to \infty} u_n = 0 \tag{2.16}$$

La proprietà espressa da (2.16) prende il nome di *assoluta stabilità.*

Supponendo che nella (2.15) λ sia un numero complesso a parte reale negativa è possibile definire la *regione di assoluta stabilità* del metodo come

l'insieme dei valori complessi $h\lambda$ per i quali il metodo è assolutamente stabile. In questa regione il metodo, essendo zero-stabile, è in grado di controllare l'errore di perturbazione del problema.

Se la regione di assoluta stabilità comprende il semipiano reale negativo il metodo si dice *A-stabile*.

È possibile definire anche l'*A(α)-stabilità* in maniera simile all'A-stabilità: un metodo è A(α)-stabile se la sua regione di stabilità assoluta comprende la parte di piano complesso $\mathcal{V}_\alpha := \{\rho e^{i\theta} : \rho > 0, |\theta + \pi| < \alpha\}$. [7]

2.4 Metodi ad un passo

I metodi ad un passo permettono di calcolare un'approssimazione della soluzione del problema ai valori iniziali al tempo t_{n+1} semplicemente conoscendo il valore della soluzione al tempo t_n.

Il metodo più semplice è il metodo di Eulero esplicito e si ottiene approssimando la derivata esatta $y'(t_n)$ con il rapporto incrementale $(\delta_+ f)(\bar{t}) = [f(\bar{t}+h) - f(\bar{t})]/h$. Si ottiene il seguente metodo convergente con ordine 1:

$$u_{n+1} = u_n + hf(t_n, u_n) \qquad n = 0, ..., N_h - 1 \tag{2.17}$$

Approssimando la derivata $y'(t_n)$ con il rapporto incrementale $(\delta_- f)(\bar{t}) = [f(\bar{t}) - f(\bar{t} - h)]/h$ si ottiene il metodo di Eulero implicito, anch'esso di ordine 1:

$$u_{n+1} = u_n + hf(t_{n+1}, u_{n+1}) \qquad n = 0, ..., N_h - 1 \tag{2.18}$$

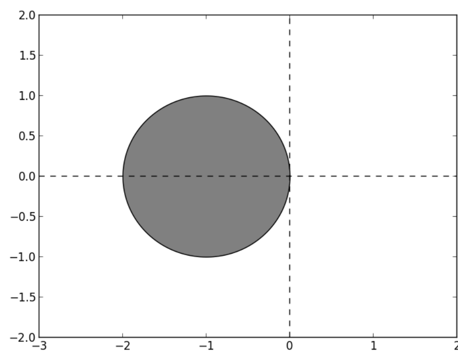
Sommando membro a membro i due metodi di Eulero è possibile ottenere un terzo metodo, il metodo del trapezio o di Crank-Nicolson:

$$u_{n+1} = u_n + \frac{h}{2}\left[f(t_n, u_n) + f(t_{n+1}, u_{n+1})\right] \qquad n = 0, ..., N_h - 1 \tag{2.19}$$

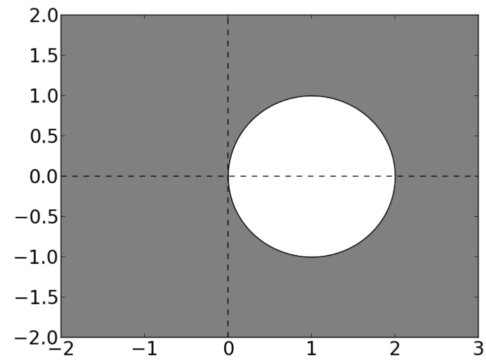
Anche questo metodo è implicito ma converge rispetto ad h con ordine 2.

Tutti i precedenti metodi possono essere riassunti nel θ-metodo, rispettivamente per $\theta = 1$, $\theta = 0$, $\theta = 1/2$:

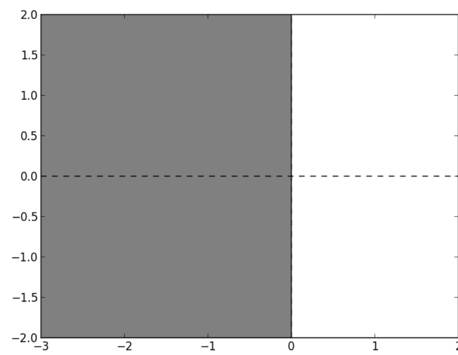
$$u_{n+1} = u_n + h\left[\theta f(t_n, u_n) + (1-\theta) f(t_{n+1}, u_{n+1})\right] \qquad \theta \in [0, 1] \tag{2.20}$$



(a) Eulero esplicito



(b) Eulero implicito



(c) Crank-Nicolson

Figura 2.1: Regioni di stabilità dei metodi ad un passo

2.5 Metodi multistep

I metodi multistep (o multipasso) consentono di raggiungere un ordine di accuratezza più elevato dei metodi ad un passo attraverso la valutazione di più valori precedenti nel calcolo del valore successivo. Ad esempio, in un metodo a 3 passi sono coinvolti nella valutazione di u_{n+1} i valori u_n, u_{n-1} e u_{n-2}.

La soluzione esatta di un'equazione del primo ordine è determinata univocamente, vista la lipschitzianità della funzione, da un'unica condizione iniziale, pertanto non è possibile pretendere che dipenda da s condizioni precedenti. Quando si passa alla soluzione numerica però la situazione cambia e i valori passati possono essere impiegati per aumentare l'accuratezza della soluzione.

Un generico metodo multistep a s passi è rappresentabile nella forma

$$\sum_{m=0}^{s} a_m u_{n+m} = h \sum_{m=0}^{s} b_m f(t_{n+m}, u_{n+m}), \qquad n = 0, 1, ... \tag{2.21}$$

dove i coefficienti a_m e b_m sono noti e indipendenti da h (solitamente si normalizza ponendo $a_s = 1$).
Il metodo multistep può essere caratterizzato dai due polinomi

$$\rho(w) := \sum_{m=0}^{s} a_m w^m \qquad \text{e} \qquad \sigma(w) := \sum_{m=0}^{s} b_m w^s \tag{2.22}$$

ed è di ordine $p \geq 1$ se e solo se esiste $c \neq 0$ tale che

$$\rho(w) - \sigma(w)\ln(w) = c(w-1)^{p+1} + \mathcal{O}(|w-1|^{p+2}), \qquad w \to 1 \tag{2.23}$$

I metodi *Adams* sono una famiglia di metodi multistep che hanno l'espressione

$$u_{n+1} = u_n + h \sum_{j=-1}^{p} b_j f_{t_{n-j}, u_{n-j}} \tag{2.24}$$

e sono basati sull'interpolazione polinomiale di Lagrange. Se l'interpolazione viene effettuata sui nodi $t_n, t_{n-1}, ..., t_{n-p}$ ($b_{-1} = 0$) lo schema risultante è esplicito e il metodo viene detto di *Adams-Bashforth*, se l'interpolazione viene effettuata su $t_{n+1}, t_n, ..., t_{n-p}$ ($b_{-1} \neq 0$) lo schema è implicito e il metodo è detto di *Adams-Moulton*.

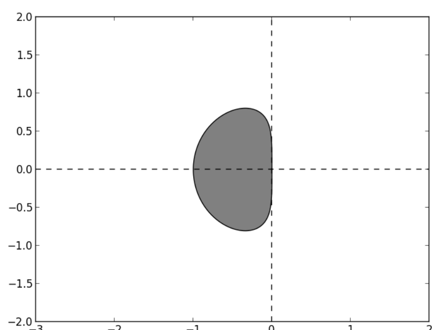
È da notare che i metodi a q passi di Adams-Bashforth generalmente sono di ordine q, quelli di Adams-Moulton di ordine $q + 1$.

Il teorema di equivalenza di Dahlquist [7] afferma che, supponendo che l'errore nei valori iniziali tenda a zero per $h \to 0^+$, il metodo multistep è convergente se e solo se è di ordine $p \geq 1$ e il polinomio ρ soddisfa la condizione delle radici.

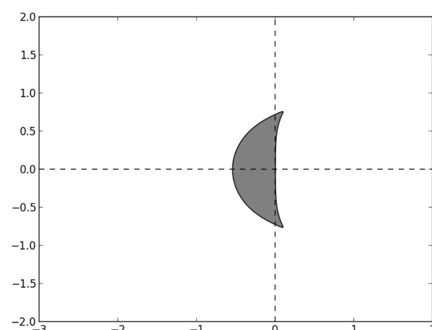
Da ciò si ricava la *prima barriera di Dahlquist*, che stabilisce in $2\lfloor(s+2)/2\rfloor$ l'ordine massimo di un metodo implicito convergente a s passi e in s l'ordine massimo di un metodo convergente esplicito.

La *seconda barriera di Dahlquist* impone un'ulteriore limitazione: non esistono metodi multistep A-stabili di ordine strettamente maggiore a 2.

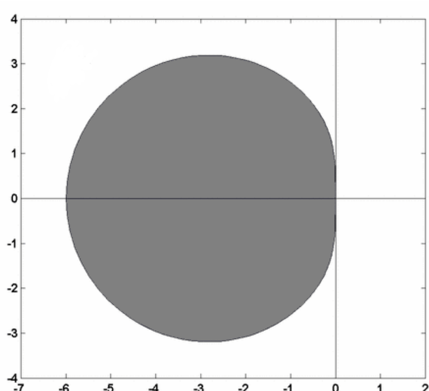
Le librerie SUNDIALS propongono un solutore che implementa i metodi Adams-Moulton di ordine da 1 a 12. Questi metodi, come si vedrà in seguito alla sezione 2.7, non sono indicati per la soluzione di problemi *stiff*. Il solutore implementato in tali librerie assicura comunque la convergenza perchè riduce l'ordine fino ad arrivare ai metodi a 1 o 2 passi che sono A-stabili.



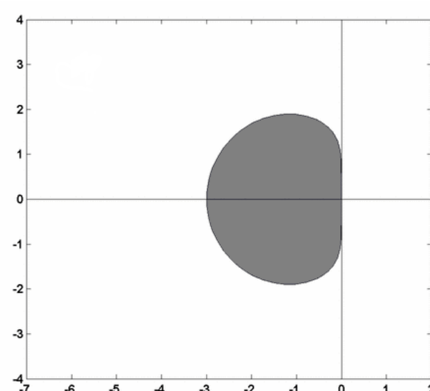
(a) Adams-Bashforth 2 passi



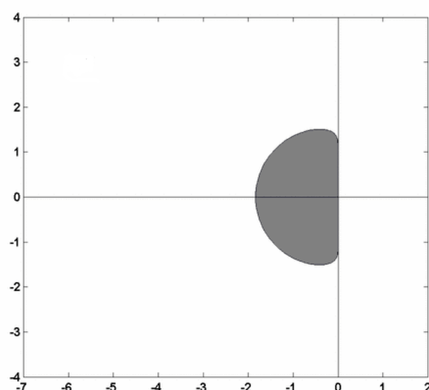
(b) Adams-Bashforth 3 passi



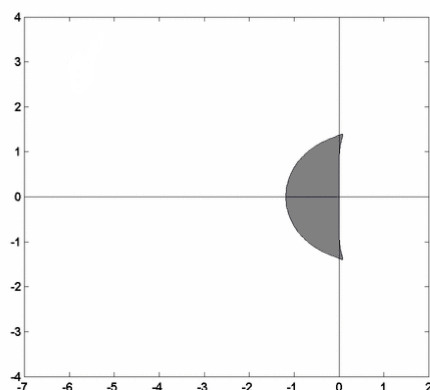
(c) Adams-Moulton 3 passi



(d) Adams-Moulton 4 passi



(e) Adams-Moulton 5 passi



(f) Adams-Moulton 6 passi

Figura 2.2: Regioni di stabilità dei metodi multistep. I metodi Adams-Bashforth ad 1 passo e i metodi Adams-Moulton a 1 e 2 passi coincidono rispettivamente con i metodi di Eulero esplicito, Eulero implicito e Crank-Nicolson. Le loro regioni di stabilità sono riportate in figura 2.1.

2.6 Metodi BDF

Quella delle *Backward Differentiation Formula* (BDF) è un'altra famiglia di metodi multistep impliciti basata sull'approssimazione della derivata $y'(t_{n+1})$ con la derivata prima del polinomio interpolatore di y di grado $p+1$ nei $p+2$ nodi $t_{n+1}, t_n, ..., t_{n-p}, \quad p \geq 0$. L'interesse a questi metodi è dovuto al loro ruolo nella risoluzione dei problemi *stiff*, che verranno presentati nella sezione 2.7.

Un metodo multistep a s passi di ordine s è una BDF se $\sigma(w) = bw^s$ per qualche $b \in \mathbb{R} \setminus \{0\}$. Avendo solo il termine di grado massimo nel polinomio σ, si ottengono schemi del tipo:

$$u_{n+1} = \sum_{j=0}^{p} a_j u_{n-j} + hbf(t_{n+1}, u_{n+1}), \qquad b \neq 0 \tag{2.25}$$

Si dimostra (cfr. [7]) che la definizione di BDF implica:

$$b = \left(\sum_{m=1}^{s} \frac{1}{m}\right) \qquad \text{e} \qquad \rho(w) = b \sum_{m=1}^{s} \frac{1}{m} w^{s-m}(w-1)^m \tag{2.26}$$

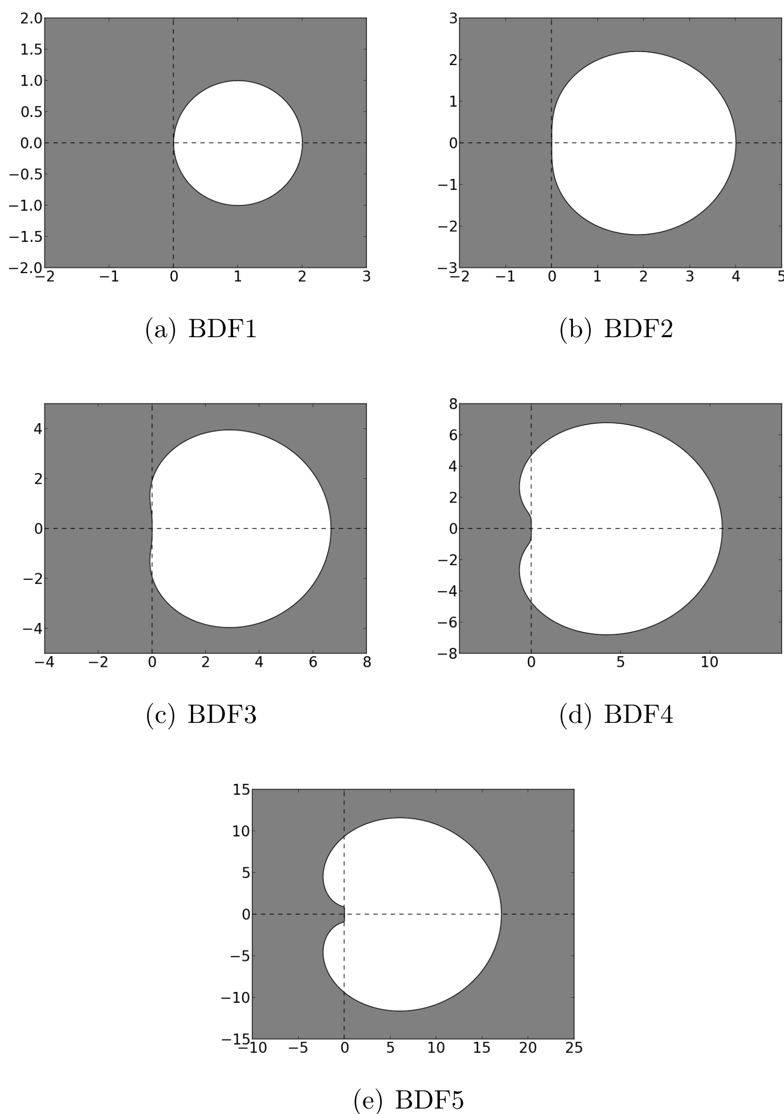
Tabella 2.1: Coefficienti formule BDF

s	b	a_1	a_2	a_3	a_4	a_5
1	1	1				
2	$\frac{2}{3}$	$\frac{4}{3}$	$-\frac{1}{3}$			
3	$\frac{6}{11}$	$\frac{18}{11}$	$-\frac{9}{11}$	$\frac{2}{11}$		
4	$\frac{12}{25}$	$\frac{48}{25}$	$-\frac{36}{25}$	$\frac{16}{25}$	$-\frac{3}{25}$	
5	$\frac{60}{137}$	$\frac{300}{137}$	$-\frac{300}{137}$	$\frac{200}{137}$	$-\frac{75}{137}$	$\frac{12}{137}$

Uno studio dell'errore di discretizzazione locale mostra che un metodo BDF è meno preciso di un metodo Adams-Moulton dello stesso ordine. Tuttavia la BDF presenta una regione di assoluta stabilità più ampia [1].
In accordo con la seconda barriera di Dahlquist, gli unici metodi A-stabili sono le BDF di ordine 1 e 2. Anche i metodi di ordine 3, 4 e 5 però presentano una regione di stabilità abbastanza estesa. Sono infatti A(α)-stabili con i seguenti angoli:

s	1	2	3	4	5
α	90°	90°	86.03°	73.35°	51.84°

Il metodo BDF a 6 passi ha una regione di stabilità molto ridotta, quelli a più di 6 passi non sono zero-stabili.



(a) BDF1

(b) BDF2

(c) BDF3

(d) BDF4

(e) BDF5

Figura 2.3: Regioni di stabilità dei metodi BDF

All'interno delle librerie CVODE di SUNDIALS è presente un solutore che implementa i metodi BDF di ordine da 1 a 5. L'ordine del metodo da utilizzare viene scelto automaticamente per assicurare la soluzione migliore, in particolare l'ordine viene ridotto se la soluzione non converge.

2.7 Problemi stiff

Il termine stiff è stato introdotto per la prima volta da Curtis e Hirschfelder nel 1952 durante lo studio di un sistema meccanico in cui alcune molle erano molto più rigide di altre. Ora con il termine *stiff* si indicano sistemi di equazioni differenziali caratterizzati da costanti di tempo molto diverse tra loro.

Tali sistemi sono stabili nel senso che la loro soluzione per $t \to \infty$ converge ad una soluzione stazionaria ma la presenza di una costante di tempo piccola costringe a sceglier un passo di integrazione molto limitato, rendendo di fatto inutilizzabile il metodo.

Si supponga di voler risolvere un'equazione differenziale del secondo ordine con il metodo di Eulero esplicito (2.17) (la trattazione seguente può essere estesa anche ad altri metodi e a equazioni di ordine superiore).

Siano λ_1 e λ_2 le radici del polinomio caratteristico associato all'equazione differenziale in esame. Di esse sia λ_1 la radice più grande in modulo e λ_2 quella più piccola.

Considerando solo l'accuratezza della soluzione si potrebbe essere portati all'utilizzo di un passo di integrazione piccolo dove la funzione varia velocemente, cioè per la durata del transitorio definito dalla costante di tempo minore (quella associata a λ_1) e all'utilizzo di un passo di integrazione più ampio successivamente, dopo l'esaurimento di tale transitorio.

Aumentando il passo ad un certo punto, precisamente quando si sceglie $h \geq -2Re(\lambda_1)/|\lambda_1|^2$, si esce dalla regione di stabilità del metodo e si ha una radice instabile. Durante la risoluzione del problema il calcolatore produce degli errori di arrotondamento, a causa dei quali prima o poi verrà assegnato un valore non nullo alla componente della soluzione associata a λ_1. Si noti che tale valore dovrebbe essere identicamente nullo, dal momento che il transitorio relativo a λ_1 si è già esaurito. Da quel momento in poi nella soluzione è presente un modo che cresce geometricamente, portando alla divergenza.

Quanto descritto è rappresentato nei grafici in figg. 2.4 e 2.5, generati dal codice in A.6.

Ne consegue che utilizzando un metodo con una regione di assoluta stabilità ridotta si ha che la componente associata alla radice più grande, che è anche quella che fornisce il suo contributo solo in un transitorio iniziale molto breve, limita fortemente il passo di integrazione anche dopo l'esaurimento del transitorio.

Per cercare di evidenziare i sistemi stiff si definisce il *quoziente di stiffness* del problema come rapporto tra la radice più grande σ e quella più piccola τ:

$$r_s = \sigma/\tau \tag{2.27}$$

Questo coefficiente però non è sempre rappresentativo della situazione reale, dal momento che il problema potrebbe avere la radice minima molto piccola in modulo e che la *stiffness* può essere influenzata dalla scelta delle condizioni iniziali, che potrebbero annullare la componente relativa ad una radice.

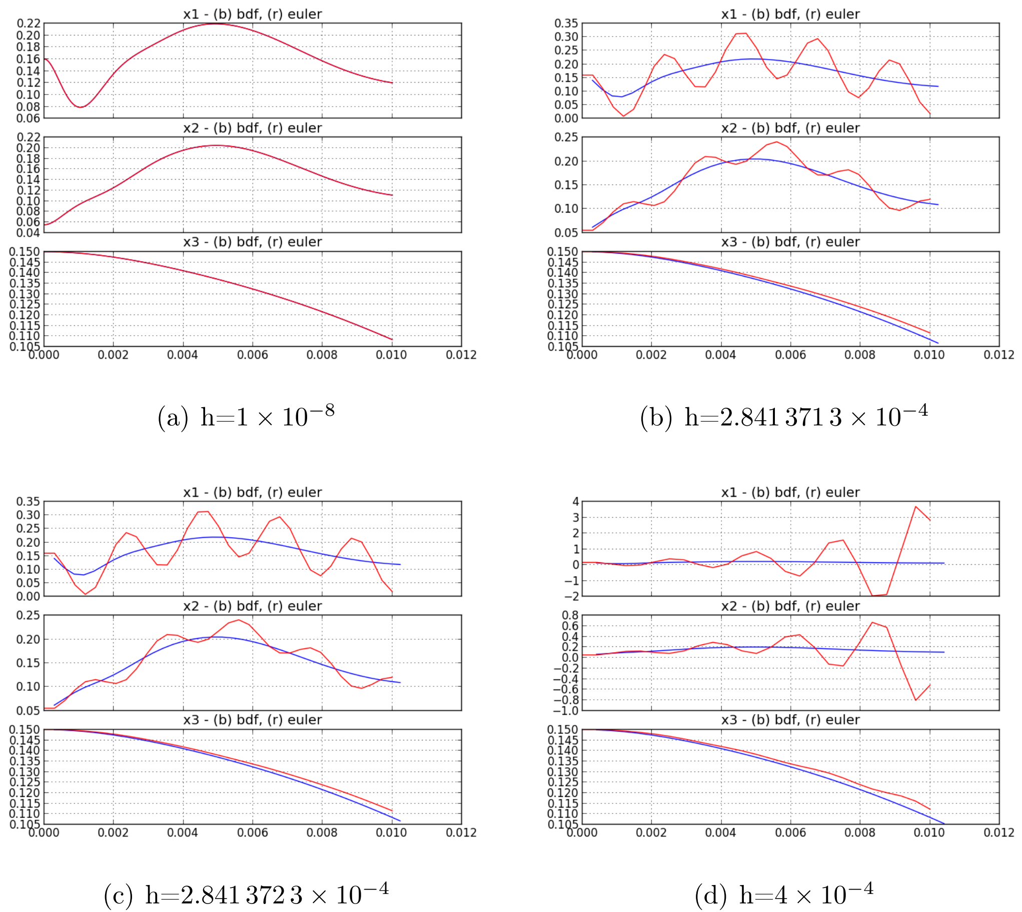
Non è quindi possibile definire esattamente un problema stiff, tuttavia una definizione che illustra bene il problema in questione è quella presentata in [9]:

Un sistema di ODE è detto *stiff* se, approssimato con un metodo numerico caratterizzato da una regione di assoluta stabilità di estensione finita, obbliga quest'ultimo, per ogni condizione iniziale per la quale il problema ammetta soluzione, ad utilizzare un passo di discretizzazione eccessivamente piccolo rispetto a quello necessario per descrivere ragionevolmente l'andamento della soluzione esatta.

Per ovviare a questo problema si rende necessario l'utilizzo di metodi con una regione di assoluta stabilità molto ampia, possibilmente infinita sul semipiano reale negativo. Si scelgono quindi metodi A-stabili o almeno A(α)-stabili.

Osservando le regioni di stabilità dei metodi Adams impliciti di ordine elevato si vede che non sono adatti a questo scopo, dal momento che essa è molto limitata.

I metodi BDF invece sono molto indicati per risolvere questo tipo di problemi perchè, come si vede dalla figura 2.3, hanno la caratteristica sopra indicata.



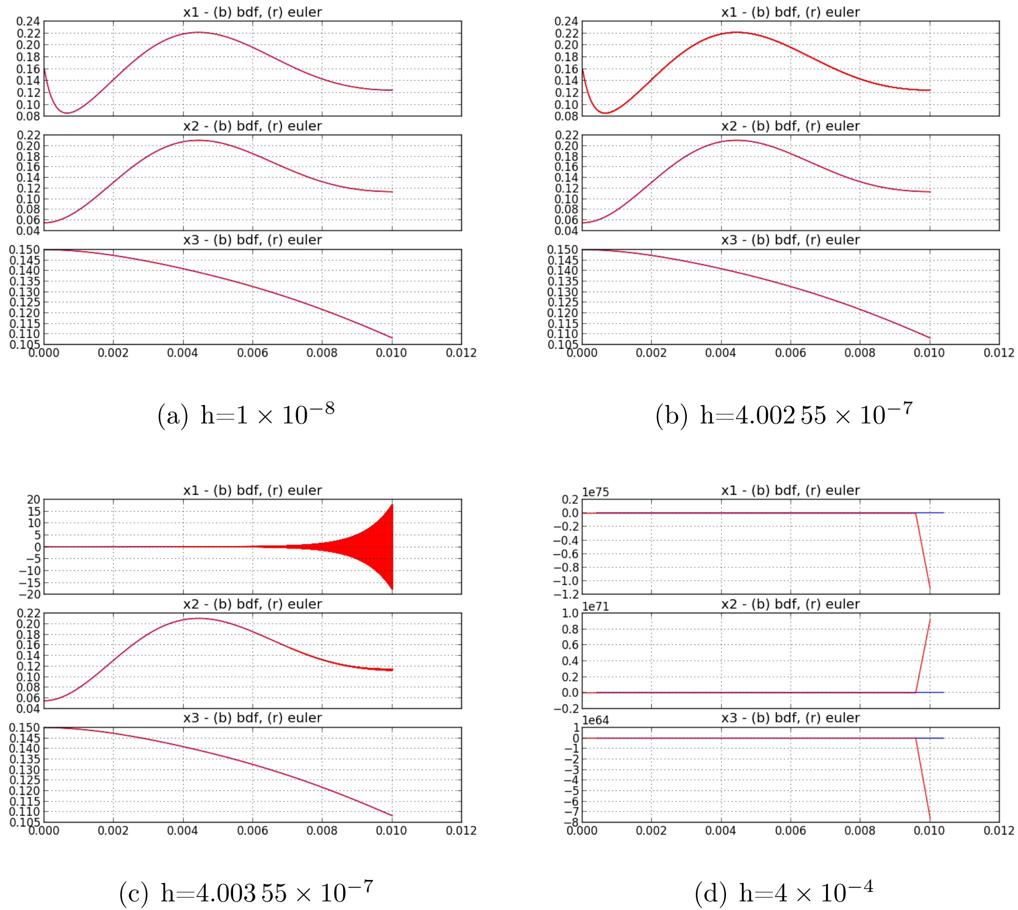
(a) h=1×10^{-8}

(b) h=$2.841\,371\,3 \times 10^{-4}$

(c) h=$2.841\,372\,3 \times 10^{-4}$

(d) h=4×10^{-4}

Figura 2.4: Confronto tra i metodi BDF implementati nelle librerie SUNDIALS e il metodo di Eulero esplicito nel caso di problemi non stiff. La condizione di assoluta stabilità del metodo di Eulero esplicito fornisce come valore critico quelli in (b), h=$2.841\,371\,3 \times 10^{-4}$, il coefficiente di stiffness è 111.37. In questo caso le due soluzioni presentano approssimativamente lo stesso andamento.



(a) h=1×10^{-8}

(b) h=$4.002\,55 \times 10^{-7}$

(c) h=$4.003\,55 \times 10^{-7}$

(d) h=4×10^{-4}

Figura 2.5: Confronto tra i metodi BDF implementati nelle librerie SUNDIALS e il metodo di Eulero esplicito nel caso di problemi stiff. La condizione di assoluta stabilità del metodo di Eulero esplicito fornisce come valore critico quello in (b), h=$4.002\,55 \times 10^{-7}$, il coefficiente di stiffness è 4.734×10^{5} In questo caso il metodo di Eulero diverge appena si esce dalla regione di stabilità.

2.8 Risoluzione di sistemi algebrici non lineari

L'utilizzo di un metodo implicito per la risoluzione di un'equazione differenziale ordinaria implica ad ogni passo la necessità di risolvere un sistema algebrico non lineare. L'alternativa a questo approccio è l'utilizzo del metodo implicito abbinato ad un metodo esplicito, cosa che però aumenta notevolmente il costo computazionale.

Applicando al problema ai valori iniziali (2.4) il generico metodo multistep (2.21) si ottiene

$$u(t_{n+s}) = hb_s f(t_{n+s}, u(t_{n+s})) + \gamma \tag{2.28}$$

dove γ rappresenta la parte esplicita (nota):

$$\gamma = h\sum_{m=0}^{s-1} b_m f(t_{n+m}, u(t_{n+m})) - \sum_{m=0}^{s-1} a_m u(t_{n+m}) \tag{2.29}$$

L'equazione (2.28) può essere risolta in vari modi, tra i quali l'iterazione funzionale e i metodi di Newton.

L'iterazione funzionale per il calcolo di $u(t_n)$ parte da una supposizione iniziale $u^{[0]}(t_n) = u(t_{n-1})$ ed è basata sull'algoritmo

$$u^{[i+1]}(t_n) = hb_s f(t_n, u^{[i]}(t_n)) + \gamma \tag{2.30}$$

Questo metodo converge se $h|b_s|\|\partial f(t_{n+s}, u_{n+s})/\partial u\| < 1$. Si ha una limitazione sul passo di discretizzazione h simile a quella imposta da alcuni metodi nella risoluzione di problemi stiff, che, anche se è di vari ordini di grandezza superiore alla precedente, rende l'applicazione di questo algoritmo ai suddetti problemi sconsigliata.

L'algoritmo di Newton propone un'iterazione alternativa nel calcolo di u_n. Il problema 2.28 può essere riformulato come

$$w = hg(w) + \gamma \tag{2.31}$$

L'algoritmo di Newton calcola il valore seguendo l'iterazione

$$w^{[i+1]} = w^{[i]} - \left[I - h\frac{\partial g(w^{[i]})}{\partial w}\right]^{-1} \left[w^{[i]} - \gamma - hg(w^{[i]})\right] \tag{2.32}$$

Questo algoritmo non presenta l'inconveniente della limitazione del passo h perchè qui la condizione di convergenza coinvolge solo la funzione f e

le sue derivate prima e seconda e generalmente è verificata per continuità nell'intorno della soluzione.

L'algoritmo di Newton converge quadraticamente ma ha due punti deboli: ad ogni passo richiede il calcolo della matrice jacobiana di f e la risoluzione di un sistema lineare. La soluzione a ciò è data dall'algoritmo di *Newton modificato*, che impiega al posto della matrice $\partial g(w^{[i]})/\partial w$ la matrice

$$J = \frac{\partial g(w^{[0]})}{\partial w} \tag{2.33}$$

che viene calcolata ad ogni passo h e mantenuta costante durante le iterazioni di Newton.

L'assenza della limitazione sul passo di discretizzazione rende questo algoritmo adatto alla soluzione dei sistemi algebrici non lineari relativi a problemi stiff, l'unico inconveniente si presenta quando la convergenza è lenta. In tal caso utilizzando l'iterazione funzionale è sufficiente diminuire il passo di discretizzazione per avere una convergenza più rapida, mentre se si utilizza l'algoritmo di Newton modificato si deve anche calcolare la nuova matrice jacobiana e si deve risolvere nuovamente il sistema $I - hJ$.

Capitolo 3

Descrizione del Sistema

Verrà preso in considerazione un sistema composto da tre masse collegate da molle e smorzatori (cfr. Fig. 3.2). Si vuole contollare la posizione della massa superiore di tale sistema applicando una forza su di essa, la cui entità verrà calcolata da microcontrollore in modo tale da seguire la traiettoria desiderata.

Il modello del sistema completo è composto da un generatore di traiettorie, da un amplificatore differenziale, da un controllore implementato in una piattaforma embedded, da un attuatore e dal modello del sistema fisico, come raffigurato in figura 3.1:

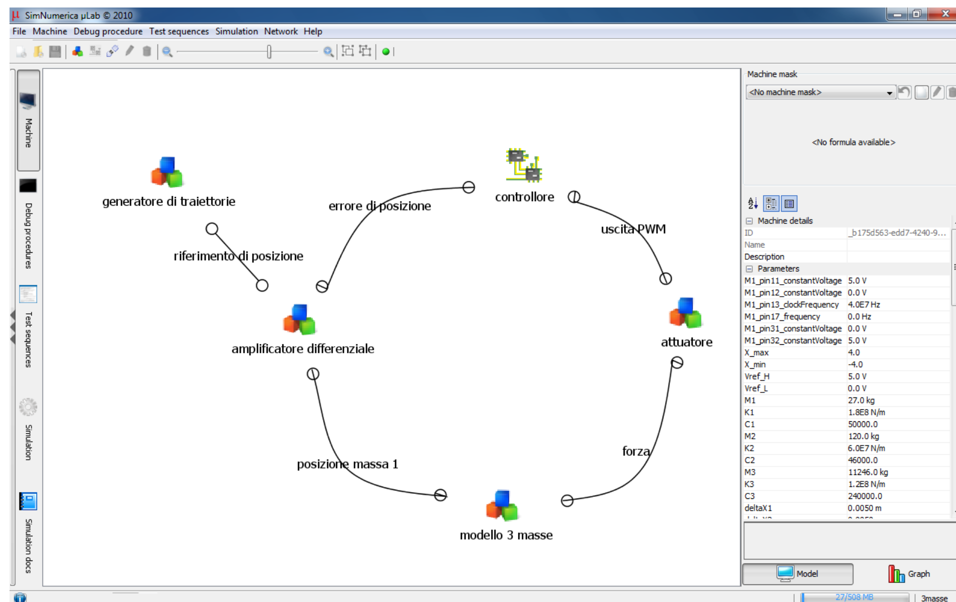


Figura 3.1: Il modello dell'intero sistema in μLab

3.1 Il sistema fisico

Il modello del sistema è rappresentato in figura 3.2 ed è composto da 3 masse, 3 molle e 3 smorzatori i cui valori sono i seguenti:

$M_1 = 27\,\mathrm{kg}$	$C_1 = 5 \times 10^4\,\mathrm{N\,s/m}$	$K_1 = 1.8 \times 10^8\,\mathrm{N/m}$
$M_2 = 120\,\mathrm{kg}$	$C_2 = 4.6 \times 10^4\,\mathrm{N\,s/m}$	$K_2 = 6 \times 10^7\,\mathrm{N/m}$
$M_3 = 1.1246 \times 10^4\,\mathrm{kg}$	$C_3 = 2.4 \times 10^5\,\mathrm{N\,s/m}$	$K_3 = 1.2 \times 10^8\,\mathrm{N/m}$

Il sistema è supposto in equilibrio quando le lunghezze delle molle sono rispettivamente $\delta x_1 = 0.005\,\mathrm{m}$, $\delta x_2 = 0.005\,\mathrm{m}$ e $\delta x_3 = 0.050\,\mathrm{m}$.

Il modello si ricava dall'equazione di Newton, $F = ma$. Applicandola alle 3 masse si ottiene il seguente sistema di equazioni differenziali:

$$\begin{cases} M_1\ddot{x_1} = C_1(\dot{x_2} - \dot{x_1}) + K_1(x_2 - x_1 + \delta x_1) + f \\ M_2\ddot{x_2} = C_2(\dot{x_3} - \dot{x_2}) + C_1(\dot{x_1} - \dot{x_2}) + K_2(x_3 - x_2 + \delta x_2) + K_1(x_1 - x_2 - \delta x_1) \\ M_3\ddot{x_3} = C_3(-\dot{x_3}) + C_2(\dot{x_2} - \dot{x_3}) + K_3(-x_3 + \delta x_3) + K_2(x_2 - x_3 - \delta x_2) \end{cases} \tag{3.1}$$

Sostituendo $y_1 = x_1, y_2 = x_2, y_3 = x_3, y_4 = \dot{x_1}, y_5 = \dot{x_2}, y_6 = \dot{x_3}$ è possibile ricavare un sistema di 6 equazioni del primo ordine, necessarie per poter applicare i metodi numerici descritti nel capitolo 2:

$$\begin{cases} \dot{y_1} = y_4 \\ \dot{y_2} = y_5 \\ \dot{y_3} = y_6 \\ \dot{y_4} = \left(C_1(y_5 - y_4) + K_1(y_2 - y_1 + \delta x_1) + f\right)/M1 \\ \dot{y_5} = \left(C_2(y_6 - y_5) + C_1(y_4 - y_5) + K_2(y_3 - y_2 + \delta x_2) + K_1(y_1 - y_2 - \delta x_1)\right)/M2 \\ \dot{y_6} = \left(C_3(-y_6) + C_2(y_5 - y_6) + K_3(-y_3 + \delta x_3) + K_2(y_2 - y_3 - \delta x_2)\right)/M3 \end{cases} \tag{3.2}$$

L'algoritmo di Newton implementato nelle librerie SUNDIALS richiede per la sua corretta esecuzione un'approssimazione della matrice jacobiana del sistema. In questo caso non è necessaria una linearizzazione di tale matrice dal momento che è costante:

$$J = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ \frac{-K1}{M1} & \frac{K1}{M1} & 0 & \frac{-C1}{M1} & \frac{C1}{M1} & 0 \\ \frac{K1}{M2} & \frac{-(K1+K2)}{M2} & \frac{K2}{M2} & \frac{C1}{M2} & \frac{-(C1+C2)}{M2} & \frac{C2}{M2} \\ 0 & \frac{K2}{M3} & \frac{-(K2+K3)}{M3} & 0 & \frac{C2}{M3} & \frac{-(C2+C3)}{M3} \end{bmatrix} \tag{3.3}$$

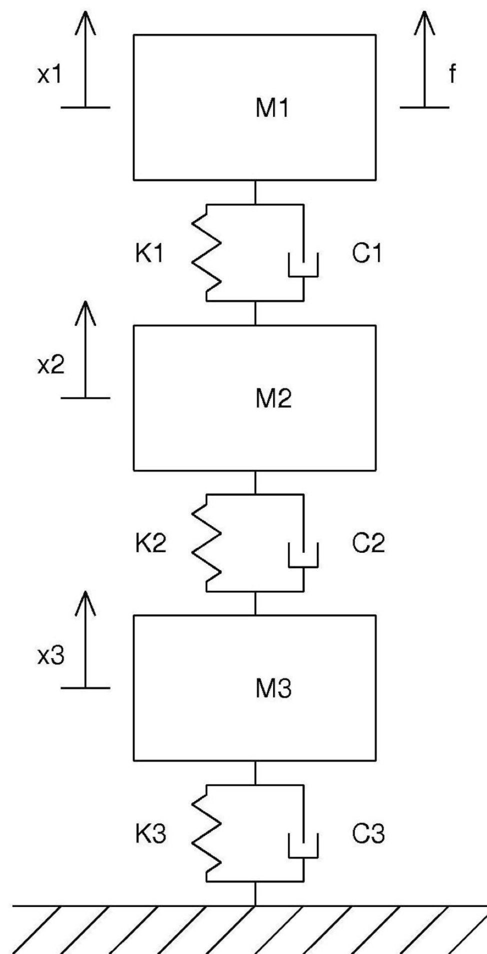


Figura 3.2: Modello a parametri concentrati del sistema preso in esame.

3.2 L'attuatore

L'attuatore è il componente che riceve in ingresso un segnale modulato in larghezza d'impulso (PWM) e fornisce in output la forza da applicare al sistema.

In base al *duty-cycle*, ossia alla frazione temporale nella quale il segnale rimane ad un livello logico alto nel periodo, l'attuatore fornisce in uscita la forza secondo una relazione di proporzionalità.

L'attuatore, essendo un componente fisico, non può fornire una forza illimitata. Sarà quindi limitato superiormente e inferiormente da due forze F_{max} e F_{min}, che si sono assunte pari a $\pm 1 \times 10^5$ N.

È possibile così, dal momento che il duty-cycle $d.c.$ è compreso tra 0 e 1, ricavare una relazione tra le due grandezze, implemenata poi nel codice in A.2:

$$F = \frac{F_{max} - F_{min}}{100} \cdot d.c. + F_{min} \tag{3.4}$$

3.3 L'amplificatore differenziale

Il controllo in retroazione richiede un confronto tra la posizione attuale della massa 1 e la posizione desiderata, in altre parole è necessario calcolare l'errore di posizione.

Dato che il segnale di errore deve essere applicato ad un pin del microcontrollore, è necessario fornire un segnale in tensione.

Il convertitore analogico-digitale che acquisisce il segnale di errore ha due valori di tensione di riferimento ed è in grado di digitalizzare valori solo all'interno del loro intervallo.

Il voler ottenere una tensione che varia linearmente con l'errore impone la limitazione massima e minima dell'errore, oltre a tali limiti si dovrà fornire in ingresso un valore di tensione costante pari alla corrispondente tensione di riferimento (se si ha un errore positivo molto grande si fornirà al pin la tensione di riferimento più alta).
In tal caso, se l'errore permane per molto tempo, il controllore richiederà all'attuatore la forza massima, cercando così di ridurre l'errore.

Per fornire una visione più ampia, all'interno del componente *amplificatore differenziale* (descritto dal codice in A.4) è stata introdotta la saturazione sull'errore in millimetri e poi è stato calcolato il valore corrispondente in volt. Si ha quindi la possibilità di valutare l'errore direttamente in millimetri dai grafici, senza dover fare la conversione dalla misura in volt.

3.4 Il generatore di traiettorie

Il generatore di traiettorie è stato introdotto con il solo scopo di simulare l'andamento desiderato per la massa superiore.
Nelle applicazioni reali spesso questo componente non esiste, il controllore acquisisce il segnale che rappresenta la posizione attuale della massa e contiene al suo interno l'andamento ideale da seguire, trasformando quindi il componente descritto come amplificatore differenziale in un semplice trasduttore.
In questa simulazione sono stati introdotti questi componenti per permettere un confronto migliore tra le posizioni attuale e di riferimento.

In questo componente sono state implementate nel codice in A.3 due traiettorie: la prima è sinusoidale e oscilla attorno al punto di equilibrio con un'ampiezza di 3 mm e una pulsazione angolare di 10 rad/s. La seconda è un segnale a gradini: si parte dalla posizione di equilibrio, dopo 20 ms la posizione di riferimento sale di 3 mm e a 400 ms scende di 2 mm.

3.5 Il controllore

Il controllore in questione è un microprocessore PIC18F4620 [8], che acquisisce il segnale di errore tramite il modulo A/D dal pin AN0, lo elabora e, in funzione della forza necessaria per l'azione di controllo, genera tramite il modulo CCP (Capture/Compare/PWM) un segnale PWM sul pin CCP1.

Al pin di uscita è connesso un *adapter*, che permette di ottenere, in uscita dalla piattaforma embedded che nel modello μLab contiene il microcontrollore, direttamente il valore del duty-cycle del segnale PWM con un errore inferiore all'1%.

Il firmware prodotto è riportato in A.5. Dopo una prima fase di calcolo di alcune costanti che non cambieranno per tutta la simulazione (utilizzate per velocizzare il calcolo all'interno del loop successivo) e di inizializzazione dei moduli necessari, si entra in un ciclo infinito, all'interno del quale è implementato l'algoritmo di controllo descritto al capitolo 4.

Per l'implementazione dell'equazione 4.9 sono necessari il valore dell'errore attuale, il valore dell'errore al passo precedente e la somma degli errori in tutti i passi precedenti.
All'inizio del ciclo si aggiornano le variabili che rappresentano gli errori passati, si attende la fine della conversione del modulo A/D e si calcola il nuovo errore. Ora è possibile calcolare il valore della forza in uscita, che però deve necessariamente essere limitata come descritto in 3.2, quindi c'è la generazione del segnale PWM.

Si noti la presenza del filtro anti wind-up, necessario per garantire errore nullo a regime anche nel caso di saturazione dell'attuatore.

Nel firmware si è scelto di convertire il valore acquisito per ottenere un errore espresso in millimetri, quindi di calcolare il valore della forza che l'attuatore deve fornire in newton e poi di trasformarlo nel valore da scrivere nel registro CCPR1L. Si è dato un significato fisico alle variabili per facilitare il confronto tra grandezze interne ed esterne alla piattaforma embedded, cosa che facilita molto l'attività di *debugging*.

Capitolo 4

Il Controllo

Il controllo del sistema che verrà descritto al capitolo 3 ha l'obiettivo di far convergere il più possibile la posizione attuale della massa 1 del sistema alla posizione di riferimento attraverso il comando di un attuatore con un segnale opportuno. Schematicamente l'intero sistema può essere rappresentato nel seguente modo:

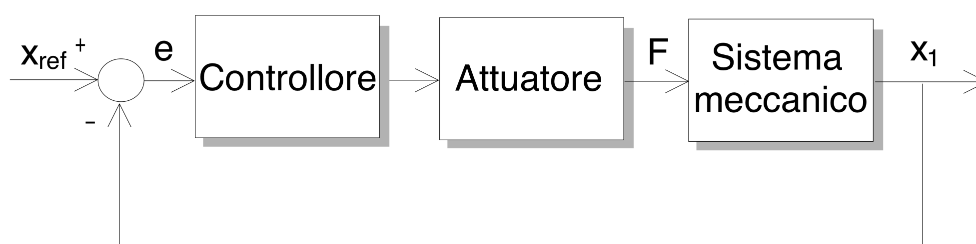


Figura 4.1: Schema del controllo in retroazione.

Il sistema meccanico è intrinsecamente stabile ma il fatto che gli attuatori possono fornire solo una forza limitata introduce una non-linearità che impedisce di utilizzare i classici metodi di analisi, rendendo necessaria la simulazione.

Il controllore che verrà implementato sarà di tipo PID. Esistono tecniche di controllo digitale più avanzate che portano a risultati migliori, ma il loro studio esula dallo scopo di questa analisi.

4.1 Il controllo PID

Il controllo PID è dato dalla somma di tre azioni che legano l'errore $e(t)$ alla variabile di controllo $F(t)$, una proporzionale, una integrale e una derivativa:

$$F(t) = F_p(t) + F_i(t) + F_d(t) \tag{4.1}$$

L'azione proporzionale lega la variabile di controllo all'errore secondo una legge del tipo

$$F_p(t) = K_p \cdot e(t) \tag{4.2}$$

Scegliendo una costante K_p grande si amplifica l'errore in ingresso, ottenendo un controllo più preciso. Gli inconvenienti compaiono quando si sceglie un valore troppo elevato.

Gli attuatori hanno un limite fisico oltre il quale saturano: si pensi ad esempio ad un motore controllato da un segnale PWM, una volta che si è raggiunto un segnale con duty cycle del 100% il motore gira alla velocità massima e non è possibile aumentarla ulteriormente.

La scelta di un coefficiente di proporzionalità troppo elevato riduce la banda di proporzionalità del controllore, definita come l'intervallo di valori nel quale l'attuatore lavora in zona lineare. In questa situazione il controllore, quando rileva un errore abbastanza elevato, risponde saturando l'attuatore e oltrepassando la grandezza di riferimento. Questo porterà successivamente ad una saturazione inversa e così via, portando a delle oscillazioni attorno al valore desiderato.

La scelta di un coefficiente di proporzionalità troppo basso porta, nel caso di un controllore senza azione integrale, ad un errore consistente ai segnali costanti, tanto più grande quanto più piccola è K_p.

L'azione integrale segue la legge

$$F_i(t) = K_i \cdot \int_0^t e(\tau) d\tau \tag{4.3}$$

Il suo obiettivo principale è quello rendere infinito il guadagno in regime continuo, unico modo per eliminare l'errore ai segnali costanti. L'integratore infatti può fornire un'uscita non nulla anche in assenza di errore in ingresso.

Il fatto che l'azione integrale rimanga costante in assenza di errore rende non trascurabile il rischio di sovraelongazione, dal momento che l'uscita dell'integratore rimane costante quando viene raggiunto l'obiettivo del controllo. Aumentano quindi le oscillazioni attorno al punto di equilibrio.

La saturazione dell'attuatore complica ulteriormente la situazione. L'azione dell'attuatore risulta più lenta di quella desiderata dal controllore e il valore dell'integrale continua ad aumentare per tutto il tempo in cui l'attuatore è saturo. Si supponga ora che l'errore inizi a calare, quando si raggiunge l'errore nullo si ha comunque un valore elevato dell'integrale che porta l'uscita ad un livello inferiore, facendo crescere l'errore nella direzione opposta.

Questo fenomeno è detto *wind up* e può essere evitato semplicemente fermando l'integrazione quando l'attuatore è in saturazione. In tale situazione il controllore fornirà comunque il valore massimo in uscita ma senza far esplodere il termine integrale. Questa tecnica prende il nome di *anti wind-up* (cfr. [4]).

L'azione derivativa è esprimibile nel seguente modo:

$$F_d(t) = K_d \cdot \dot{e}(t) \tag{4.4}$$

Essendo proporzionale alla derivata dell'errore, l'effetto del derivatore è massimo quando l'uscita attraversa la grandezza di riferimento. È utile per smorzare l'uscita, consentendo la scelta di una costante di proporzionalità più elevata. Un valore troppo elevato però crea dei picchi notevoli nell'uscita quando si ha una variazione rapida dell'errore in ingresso.

Il controllore PID ha un campo di applicazione piuttosto ampio e il suo vantaggio principale è il fatto di non richiedere un modello matematico del processo da controllare ma solamente una taratura del controllore.

Ci sono vari metodi per tarare un controllore di questo tipo, tra cui si citano il *Trial-and-Error*, le regole di Ziegler-Nichols e altri metodi basati sulla posizione dei poli (cfr. [4]).

Qui si è scelto di utilizzare il metodo *Trial-and-Error*. Si tratta di un metodo a tentativi composto da vari passi:

- Si implementa un controllo puramente proporzionale (ponendo K_i e K_d a zero) con un K_p basso, si applica un segnale di riferimento a gradino di ampiezza ridotta (per restare all'interno della banda di proporzionalità).

- Si comincia a raddoppiare la costante di proporzionalità, lo smorzamento comincia a calare e aumentano le oscillazioni. Si continua in questo modo fino al superamento del livello massimo di oscillazioni accettabili.

- Ora si aggiunge la parte derivativa, aumentando K_d finchè la risposta del sistema è abbastanza smorzata. Se necessario, a questo punto, si

può aumentare ancora un po' K_p, fino al raggiungimento di un buon compromesso. Se aggiungendo la parte derivativa il sistema diventa oscillatorio è necessario utilizzare solo un controllo di tipo PI.

- Manca adesso solo l'azione integrale. Aumentando K_i l'errore a regime tenderà a zero sempre più velocemente. Si continua a raddoppiare K_i finchè la risposta oscilla a bassa frequenza.

- Come ultimo passo, per ridurre ulteriormente le oscillazioni, si può abbassare un po' il guadagno, riducendo contemporaneamente le tre costanti che caratterizzano il controllore.

Il risultato ottimale è una risposta al gradino che oscilla a bassa frequenza attorno al valore di riferimento e lo raggiunge abbastanza rapidamente.

Da alcune simulazioni si ottengono i seguenti valori per le costanti del controllore:

$$\begin{aligned} K_p &= 2 \times 10^3 \\ K_i &= 9 \times 10^5 \\ K_d &= 4 \end{aligned} \tag{4.5}$$

4.2 Discretizzazione e algoritmo di posizione

L'implementazione del controllo descritto in un microcontrollore richiede una discretizzazione della legge di controllo del PID. Per fare ciò si punta a discretizzare singolarmente le tre componenti (cfr. [2]). La parte più semplice ovviamente è quella proporzionale, che porge

$$F_p(h) = K_p \cdot e(h) \tag{4.6}$$

Per la parte integrale si utilizza l'approssimazione di Eulero:

$$F_i(h) = K_i \cdot \sum_{i=1}^{h} e(i-1)T_c \tag{4.7}$$

Infine la parte derivativa si approssima con una differenza all'indietro

$$F_d(h) = K_d \cdot \frac{e(h) - e(h-1)}{T_c} \tag{4.8}$$

La somma di queste componenti fornisce l'algoritmo di controllo di posizione:

$$F(h) = K_p e(h) + K_i T_c \sum_{i=1}^{h} e(i-1) + K_d/T_c(e(h) - e(h-1)) \tag{4.9}$$

In queste formule si è assunto implicitamente T_c come passo di discretizzazione. Tale valore è limitato inferiormente dal tempo impiegato dal microcontrollore ad acquisire il segnale di errore e calcolare il valore di uscita. È importante limitare il più possibile il tempo T_c perchè in questo intervallo temporale non si ha il controllo su ciò che accade al sistema. In particolare se l'errore varia sensibilmente tra acquisizione e variazione dell'uscita si ha un controllo errato, che può rallentare (in alcuni casi anche impedire) la convergenza.

4.3 Risultati delle simulazioni

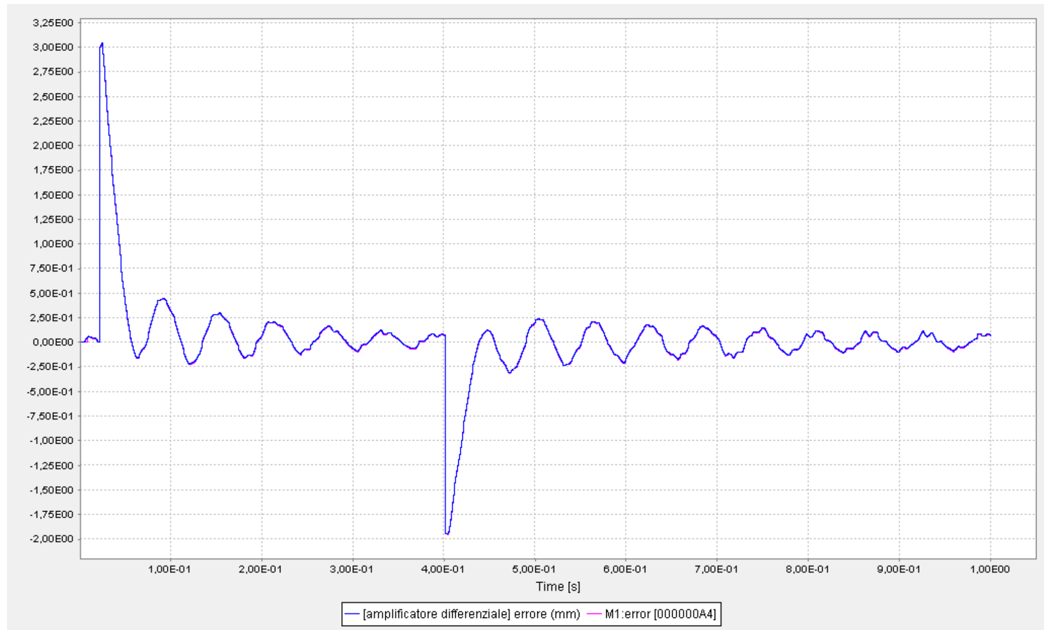
Di seguito si riportano in figg. 4.2 e 4.3 gli andamenti delle variabili più significative delle co-simulazioni effettuate.

Si nota che gli andamenti delle variabili calcolati all'interno del microcontrollore sono simili a quelli delle variabili esterne, a meno di qualche approssimazione introdotta nelle varie conversioni.

Il ritardo introdotto dal calcolo del segnale di controllo introduce uno sfasamento nel caso di riferimento sinusoidale, mentre con riferimento a gradini ha come unico effetto un picco nell'andamento dell'errore.

Figura 4.2: Grafici delle co-simulazioni effettuate in μLab con riferimento di posizione a gradini. Nella grafico (a) in rosso è raffigurato il segnale di riferimento di posizione, che indica la traiettoria che si vuol fare seguire alla massa 1, in verde la traiettoria seguita dal sistema. Nel grafico (b) sono raffigurati gli errori di posizione in millimetri. In blu l'errore calcolato dall'amplificatore differenziale, in viola il valore dell'errore calcolato all'interno del microcontrollore partendo dal risultato della conversione A/D. Nel grafico (c) sono infine raffigurati gli andamenti temporali della forza. Quella calcolata dal microcontrollore è rappresentata in verde, quella impressa dall'attuatore in rosso.

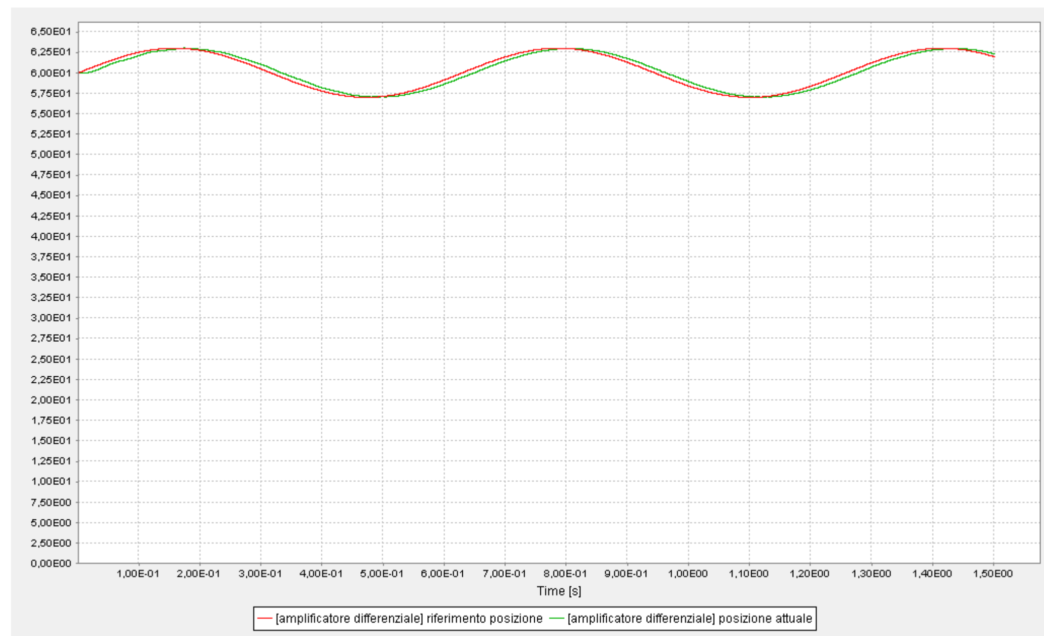
(a) Andamento della traiettoria della massa superiore.



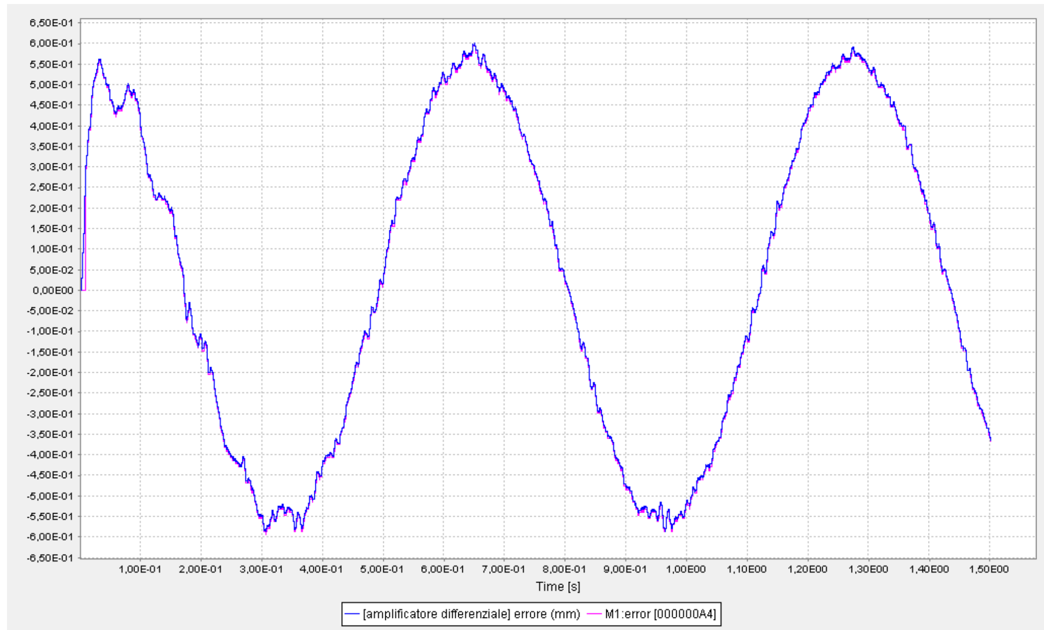
(b) Andamento dell'errore di posizione in millimetri.

(c) Andamento della forza impressa al sistema.

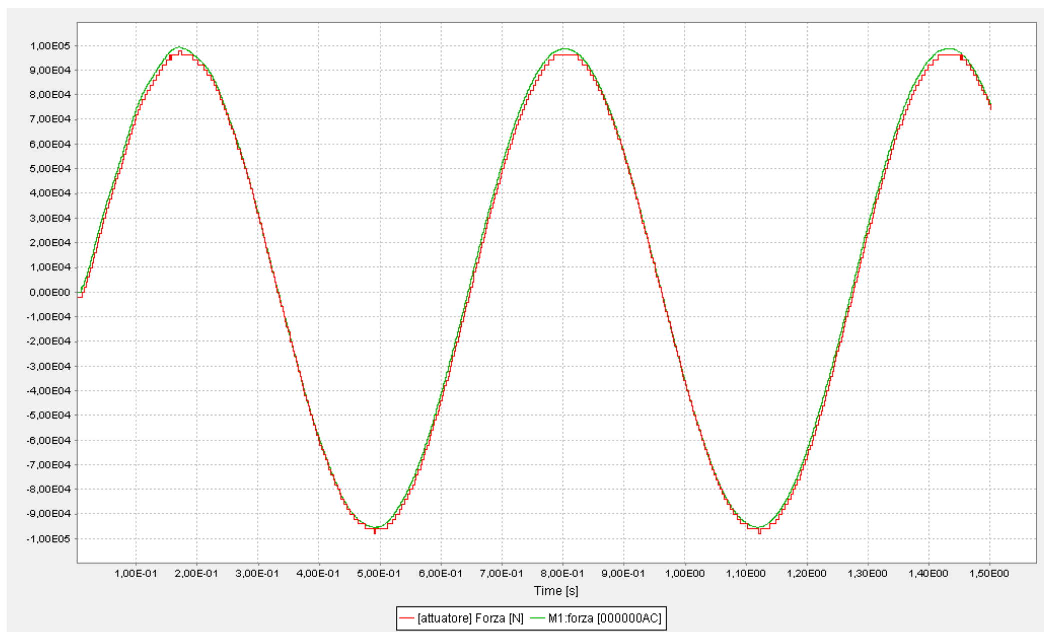
Figura 4.3: Grafici delle co-simulazioni effettuate in μLab con riferimento di posizione sinusoidale. Nella grafico (a) in rosso è raffigurato il segnale di riferimento di posizione, che indica la traiettoria che si vuol fare seguire alla massa 1, in verde la traiettoria seguita dal sistema. Nel grafico (b) sono raffigurati gli errori di posizione in millimetri. In blu l'errore calcolato dall'amplificatore differenziale, in viola il valore dell'errore calcolato all'interno del microcontrollore partendo dal risultato della conversione A/D. Nel grafico (c) sono infine raffigurati gli andamenti temporali della forza. Quella calcolata dal microcontrollore è rappresentata in verde, quella impressa dall'attuatore in rosso.



(a) Andamento del riferimento di posizione e della posizione reale della massa superiore.



(b) Andamento dell'errore di posizione in millimetri.



(c) Andamento della forza impressa al sistema.

Capitolo 5

Conclusioni

In questo elaborato si sono analizzati i vari metodi numerici disponibili per la risoluzione di sistemi di equazioni differenziali ordinarie nel caso di problemi stiff e non-stiff, concludendo che per i primi sono da preferire le formule BDF e per i secondi i metodi impliciti di Adams-Moulton, più veloci ma con una regione di stabilità ridotta.

All'interno delle co-simulazioni si è scelto di utilizzare i metodi BDF per non avere alcun tipo di vincolo sul passo di discretizzazione diverso dalla necessità di accuratezza della soluzione, indipendentemente dal fattore di stiffness. È stata utilizzata l'implementazione fornita nelle librerie CVODE della suite SUNDIALS, che si serve dell'algoritmo di Newton per risolvere il sistema derivante dall'adozione di un metodo implicito. L'obiettivo di convergenza in queste iterazioni impone variazioni non troppo veloci della traiettoria della massa superiore.

Il sistema in questione è stato controllato per seguire la traiettoria desiderata tramite l'algoritmo di controllo di posizione, che discretizza il PID. La taratura, vista l'empiricità del metodo, non è precisa ma porta ugualmente a dei risultati accettabili: si noti che in entrambi i casi la convergenza della posizione attuale a quella di riferimento è piuttosto veloce e l'errore massimo (eccetto agli istanti di applicazione dei gradini) è inferiore al 20%, nel caso di risposta al gradino anche di molto. L'algoritmo implementato può essere migliorato riducendo il tempo impiegato dal microcontrollore ad eseguire il ciclo di controllo, ad esempio eliminando la corrispondenza tra variabili fisiche esterne e variabili utilizzate per il controllo, molto utile in fase di *debugging* ma oneroso dal punto di vista computazionale. Con un tempo di controllo inferiore infatti si ottiene una convergenza più rapida. Esistono inoltre tecniche di controllo digitale e di taratura del controllore più complesse che permettono di ridurre ulteriormente l'errore.

Una co-simulazione evidenzia anche altri aspetti che in una prima ana-

lisi si tende a trascurare, quali la necessità di operare svariate conversioni tra le grandezze in gioco per adattarle ai dispositivi utilizzati. Si citano ad esempio la conversione dell'errore di posizione in volt e del segnale PWM fornito in uscita dal microcontrollore in forza, conversione operata dall'adapter seguito dall'attuatore. Ci si trova anche a trattare le non-linearità dei dispositivi, qui intese semplicemente a livello di saturazione dei vari componenti, che obbligano a prestare ancora più attenzione nel collegamento dei vari dispositivi.

Alcuni problemi si possono creare nella fase di inizializzazione del controllore. Prima di entrare nel loop di controllo infatti è necessario attivare le varie periferiche. Prima di attivare il modulo CCP che fornisce il segnale di controllo all'attuatore si ha in uscita dal modulo PWM un segnale costante con duty cycle nullo, che in questa applicazione comanda il sistema applicando la massima forza negativa. Per ovviare a questo problema è stata inserita un'istruzione condizionale che fornisce errore nullo in ingresso (e quindi uscita nulla dal momento che l'integratore è inizializzato a zero) per i primi 8 ms, in futuro si può risolvere il problema in altro modo, ad esempio considerando il duty cycle come indicatore del modulo della forza da applicare e utilizzando un altro pin della piattaforma embedded per rappresentare il segno (e quindi la direzione) della forza.

Si conclude notando che la co-simulazione permette non solo lo studio complessivo dell'intero sistema, ma anche il miglioramento del sistema di controllo attraverso la ricerca di soluzioni alternative.

Appendice A

Codice sorgente

A.1 Modello Python 3 masse

```
# <YOUR CODE (START)>: put here additional "import" that you need
from pysundials import cvode
import ctypes

#========== PARAMETRI CVODE
abstol = 1e-6   # tolleranza assoluta
reltol = 1e-6   # tolleranza relativa
N = 6           # dimensione problema
numsteps = 1500 # numero massimo step

#========== Equazioni differenziali del sistema
def f(t,y,ydot,f_data):
    global forza
    ydot[0] = y[3]
    ydot[1] = y[4]
    ydot[2] = y[5]
    ydot[3] = (C1*(y[4]-y[3]) + K1*(y[1]-y[0]+deltaX1) + forza)/M1
    ydot[4] = (C2*(y[5]-y[4]) + C1*(y[3]-y[4]) + K2*(y[2]-y[1]+deltaX2) + K1
        *(y[0]-y[1]-deltaX1))/M2
    ydot[5] = (C3*(-y[5]) + C2*(y[4]-y[5]) + K3*(   -y[2]+deltaX3) + K2*(y
        [1]-y[2]-deltaX2))/M3
    return 0

#========== Matrice Jacobiana di 'f'
def Jac(N, J, t, y, fy, jav_data, tmp1, tmp2, tmp3):
    J[0][3] = 1
    J[1][4] = 1
    J[2][5] = 1

    J[3][0] = -K1/M1
    J[3][1] =  K1/M1
    J[3][3] = -C1/M1
    J[3][4] =  C1/M1

    J[4][0] =  K1/M2
    J[4][1] = -(K1+K2)/M2
    J[4][2] =  K2/M2
    J[4][3] =  C1/M2
    J[4][4] = -(C1+C2)/M2
```

```
38      J[4][5] =  C2/M2
39
40      J[5][1] =  K2/M3
41      J[5][2] = -(K2+K3)/M3
42      J[5][4] =  C2/M3
43      J[5][5] = -(C2+C3)/M3
44      return 0
45
46 # <YOUR CODE (END)>
47
48 def solve(component, context, external_data):
49      # <YOUR CODE (START)>: put here the actual solution of the component
            model.
50      global forza, M1, M2, M3, C1, C2, C3, K1, K2, K3, deltaX1, deltaX2,
            deltaX3, cvode_mem1
51
52 #========== Carico i parametri del sistema
53      M1 = parameters['M1']             # Kg
54      M2 = parameters['M2']             # Kg
55      M3 = parameters['M3']             # Kg
56      K1 = parameters['K1']             # N/m
57      K2 = parameters['K2']             # N/m
58      K3 = parameters['K3']             # N/m
59      C1 = parameters['C1']             # N*s/m
60      C2 = parameters['C2']             # N*s/m
61      C3 = parameters['C3']             # N*s/m
62      deltaX1 = parameters['deltaX1']  # m
63      deltaX2 = parameters['deltaX2']  # m
64      deltaX3 = parameters['deltaX3']  # m
65
66      forza = variables['forza']
67      x1 = variables['x1']/1000.0 # conversione da millimetri a metri
68      x2 = variables['x2']/1000.0
69      x3 = variables['x3']/1000.0
70      dot_x1 = variables['dot_x1']
71      dot_x2 = variables['dot_x2']
72      dot_x3 = variables['dot_x3']
73      y = cvode.NVector([x1, x2, x3, dot_x1, dot_x2, dot_x3])
74      t = cvode.realtype(globalTime)
75      tout = t.value + samplingPeriod
76
77 #========== Inizializzazione del solutore
78      if globalTime==0:
79          cvode_mem1 = cvode.CVodeCreate(cvode.CV_BDF, cvode.CV_NEWTON)
80          cvode.CVodeMalloc(cvode_mem1, f, t, y, cvode.CV_SS, reltol, abstol)
81          cvode.CVDense(cvode_mem1, N)
82          cvode.CVDenseSetJacFn(cvode_mem1, Jac, None)
83          cvode.CVodeSetStabLimDet(cvode_mem1, True) # riduce ordine se a
                limite stabilita'
84          cvode.CVodeSetMaxNumSteps(cvode_mem1, numsteps)
85
86 #========== Calcolo della soluzione del sistema
87      flag = cvode.CVode(cvode_mem1, tout, y, ctypes.byref(t), cvode.CV_NORMAL
            )
88      if flag != cvode.CV_SUCCESS:
89          raise Exception('PySUNDIALS error!')
90
91 #========== Scrittura sulle variabili del modello
92      variables['x1'] = y[0]*1000
93      variables['x2'] = y[1]*1000
94      variables['x3'] = y[2]*1000
95      variables['dot_x1'] = y[3]
```

```
    variables['dot_x2'] = y[4]
    variables['dot_x3'] = y[5]
    # <YOUR CODE (END)>
```

A.2 Modello Python attuatore

```
def solve(component, context, external_data):
    # <YOUR CODE (START)>: put here the actual solution of the component
        model.
    F_max = parameters['Forza massima']
    F_min = parameters['Forza minima']
    PWM = variables['Duty cycle PWM']

    Forza = (F_max - F_min)/100.0*PWM + F_min

    variables['Forza'] = Forza
    # <YOUR CODE (END)>
```

A.3 Modello Python generatore di traiettorie

```
# <YOUR CODE (START)>: put here additional "import" that you need
import math

__SIN__   = 1
__STEPS__ = 2
# <YOUR CODE (END)>

def solve(component, context, external_data):
    # <YOUR CODE (START)>: put here the actual solution of the component
        model.
    mode = parameters['mode']
    eq = parameters['posizione equilibrio x1']
    ampiezza = parameters['ampiezza']
    omega = parameters['omega']
    delay1 = parameters['ritardo gradino 1']
    delay2 = parameters['ritardo gradino 2']
    ampiezza2 = parameters['ampiezza gradino 2']

    if mode == __SIN__:
    # riferimento di posizione di x1 sinusoidale
        x1 = eq + ampiezza * math.sin( omega * globalTime )
    elif mode == __STEPS__:
    # riferimento di posizione di x1 a 2 gradini
        x1 = eq
        if globalTime > delay1:
            x1 = x1 + ampiezza
        if globalTime > delay2:
            x1 = x1 + ampiezza2

    variables['posizione x1'] = x1
    # <YOUR CODE (END)>
```

A.4 Modello Python amplificatore differenziale

```
def solve(component, context, external_data):
    # <YOUR CODE (START)>: put here the actual solution of the component
        model.
    riferimento = variables['riferimento posizione']
    attuale = variables['posizione attuale']
    X_min = parameters['X_min']
    X_max = parameters['X_max']
    Vref_H = parameters['Vref_H']
    Vref_L = parameters['Vref_L']

    error_x = riferimento - attuale
    # limito l'errore superiormente e inferiormente
    if error_x < X_min:
        error_x = X_min
    elif error_x > X_max:
        error_x = X_max

    # conversione da millimetri a volt
    error_out = (error_x - X_min)/(X_max - X_min)*(Vref_H - Vref_L)

    # errore nullo durante l'inizializzazione del pic
    # dopo 8e-3 inizia il controllo
    if globalTime < 8e-3:
        error_out = 2.5

    variables['errore (V)'] = error_out
    variables['errore (mm)'] = error_x
    # <YOUR CODE (END)>
```

A.5 Firmware del microcontrollore in C

```
#include<p18f4620.h>

// use external clock signal
#pragma config OSC = EC

//================================= PARAMETRI PID
float Kp = 2e3;
float Ki = 9e5;
float Kd = 4e0;
float Tc = 8e-4;

//================================= PARAMETRI INPUT E OUTPUT
float MAXOUT=1.0e5;
float MINOUT=-1.0e5;
float MAXIN=4.0; // mm
float MININ=-4.0; // mm

float somma_e=0;    // somma di tutti gli errori passati
float error=0;      // errore acquisito dal modulo A/D
float lasterror=0;  // errore al passo precedente
float forza;        // forza che deve fornire l'attuatore
float A,B,C,D;      // costanti per calcolo uscita

/*
 * Routine inizializzazione periferiche
 */
```

```
27 void init(void) {
28     /*
29      * Costanti per permettere un calcolo piİ rapido
30      * dell'uscita
31      */
32     A = (Kp+Kd/Tc);
33     B = Ki*Tc;
34     C = -Kd/Tc;
35     D = (255.0/(MAXOUT-MINOUT));
36 
37     /*
38      * Imposto il pin RA0 (AN0) come pin di input
39      * e il pin RC2 (CCP1) come pin di output
40      */
41     TRISAbits.RA0 = 1;
42     TRISCbits.RC2 = 0;
43 
44     /*
45      * Impostazioni e avvio del modulo A/D:
46      * tensioni di riferimento Vdd-Vss,
47      * canale 0, risultato allineato a destra,
48      * clock Fosc/32, ritardo acquisizione 2*Tad
49      */
50     ADCON1bits.PCFG = 0b1110;
51     ADCON1bits.VCFG = 0b00;
52     ADCON0bits.CHS = 0;
53     ADCON2bits.ACQT = 0b001;
54     ADCON2bits.ADCS = 0b010;
55     ADCON2bits.ADFM = 1;
56     ADCON0bits.ADON = 1;
57 
58     /*
59      * Impostazione e avvio del modulo Timer2:
60      * clock Fosc
61      */
62     T2CONbits.T2CKPS = 0;
63     T2CONbits.TMR2ON = 1;
64 
65     /*
66      * Impostazione del modulo CCP1:
67      * segnale PWM con periodo 6.4e-6 s
68      * e duty-cycle iniziale 50%
69      */
70     PR2 = 255;
71     CCPR1L = 128;
72     CCP1CONbits.DC1B = 0;
73     CCP1CONbits.CCP1M = 0b1100;
74 }
75 
76 void main(void) {
77     init();
78     while(1) {
79         // Avvio conversione A/D
80         ADCON0bits.GO = 1;
81 
82         // Aggiorno i valori di errore passati
83         lasterror = error;
84         somma_e = somma_e + lasterror;
85 
86         // Attensa fine conversione ADC
87         while(ADCON0bits.GO==1);
```

```
88
89          // Acquisizione nuovo errore
90          error = ((ADRESH*256+ADRESL)/1024.0)*(MAXIN-MININ) + MININ;
91
92          // Calcolo uscita
93          forza = A*error + B*somma_e + C*lasterror;
94
95          // Anti wind-up
96          if (forza>MAXOUT){
97              forza = MAXOUT;
98              somma_e -= lasterror;
99          }
100         else if (forza<MINOUT){
101             forza = MINOUT;
102             somma_e -= lasterror;
103         }
104
105         // Conversione forza a 0:255
106         CCPR1L = (unsigned char)(D*(forza-MINOUT));
107     }
108 }
```

A.6 Codice Python per esempio problemi stiff

```
1  from pysundials import cvode
2  import ctypes
3  import pylab as p
4  import numpy as np
5
6  #========================================= PARAMETRI DEL SISTEMA
7  M1 = 27.0              # Kg
8  M2 = 120.0             # Kg
9  M3 = 1.1246e4          # Kg
10 K1 = 18.0e7            # N/m
11 K2 = 6.0e7             # N/m
12 K3 = 1.2e8             # N/m
13 C1 = 5.0e4             # N*s/m
14 C2 = 4.6e4             # N*s/m
15 C3 = 2.4e5             # N*s/m
16 deltaXg = 0.005        # m
17 deltaXb = 0.005        # m
18 deltaXs = 0.05         # m
19 forza = 1e6            # N
20
21 #========================================= MODIFICHE AI PARAMETRI DEL SISTEMA
22 if (True): # per renderlo stiff
23     M1 = 0.01
24
25 #========================================= PARAMETRI DI SIMULAZIONE
26 tstart = 0.0
27 tend   = 1.0e-2
28 h = 1e-7
29 condizioni_iniziali = cvode.NVector([0.16, 0.155, 0.15, 0, 0, 0])
30 abstol = 1e-6     # tolleranza assoluta
31 reltol = 1e-6     # tolleranza relativa
32 N = 6             # dimensione problema
33 numsteps = 500  # numero massimo step
34
35 m = np.array([[0, 0, 0, 1, 0, 0],
36               [0, 0, 0, 0, 1, 0],
```

```
37                  [0, 0, 0, 0, 0, 1],
38                  [-K1/M1, K1/M1, 0, -C1/M1, C1/M1, 0],
39                  [K1/M2, -(K1+K2)/M2, K2/M2, C1/M2, -(C1+C2)/M2, C2/M2],
40                  [0, K2/M3, -(K2+K3)/M3, 0, C2/M3, -(C2+C3)/M3]])
41
42 # Ricava gli autovalori e restituisce il fattore di stiffness
43 def stiffness():
44     autovalori, autovettori = np.linalg.eig(m)
45     return max(np.abs(np.real(autovalori)))/min(np.abs(np.real(autovalori)))
46
47
48 # Calcola il valore di 'h' critico per il metodo di Eulero esplicito
49 def calcoloHEuler():
50     autovalori, autovettori = np.linalg.eig(m)
51     hcr = np.min(-2*np.real(autovalori)/(np.abs(autovalori)**2))
52
53     if(False): # stampa gli autovalori e la regione di convergenza
54         w=np.exp(1j*np.linspace(0,2*np.pi,200))
55         p.figure(); p.hold(True)
56         p.plot(p.real(hcr*autovalori), p.imag(hcr*autovalori))
57         p.plot(p.real(w)-1, p.imag(w))
58         p.show()
59     return hcr
60
61
62 # Equazioni differenziali del sistema
63 def f(t,y,ydot,f_data):
64     ydot[0] = y[3]
65     ydot[1] = y[4]
66     ydot[2] = y[5]
67     ydot[3] = (C1*(y[4]-y[3]) + K1*(y[1]-y[0]+deltaXg) + forza)/M1
68     ydot[4] = (C2*(y[5]-y[4]) + C1*(y[3]-y[4]) + K2*(y[2]-y[1]+deltaXb) + K1
        *(y[0]-y[1]-deltaXg))/M2
69     ydot[5] = (C3*(-y[5]) + C2*(y[4]-y[5]) + K3*(-y[2]+deltaXs) + K2*(y[1]-y
        [2]-deltaXb))/M3
70     return 0
71
72
73 # Matrice Jacobiana di 'f'
74 def Jac(N, J, t, y, fy, jav_data, tmp1, tmp2, tmp3):
75     J[0][3] = 1
76     J[1][4] = 1
77     J[2][5] = 1
78
79     J[3][0] = -K1/M1
80     J[3][1] =  K1/M1
81     J[3][3] = -C1/M1
82     J[3][4] =  C1/M1
83
84     J[4][0] =  K1/M2
85     J[4][1] = -(K1+K2)/M2
86     J[4][2] =  K2/M2
87     J[4][3] =  C1/M2
88     J[4][4] = -(C1+C2)/M2
89     J[4][5] =  C2/M2
90
91     J[5][1] =  K2/M3
92     J[5][2] = -(K2+K3)/M3
93     J[5][4] =  C2/M3
94     J[5][5] = -(C2+C3)/M3
95     return 0
96
```

```
97
98 def solveOdeBDF():
99     tt = []; y0 = []; y1 = []; y2 = [] # per la stampa
100    t = cvode.realtype(tstart)
101    tout = tstart
102    y = cvode.NVector([0.16, 0.055, 0.15, 0, 0, 0]) # condizioni iniziali
103
104    cvode_mem1 = cvode.CVodeCreate(cvode.CV_BDF, cvode.CV_NEWTON)
105    cvode.CVodeMalloc(cvode_mem1, f, 0.0, y, cvode.CV_SS, reltol, abstol)
106    cvode.CVDense(cvode_mem1, N)
107    cvode.CVDenseSetJacFn(cvode_mem1, Jac, None)
108    cvode.CVodeSetStabLimDet(cvode_mem1, True) # riduce ordine se a limite
           stabilita'
109    cvode.CVodeSetMaxNumSteps(cvode_mem1, numsteps)
110
111    while tout < tend:
112        tout = t.value + h
113        flag = cvode.CVode(cvode_mem1, tout, y, ctypes.byref(t), cvode.
               CV_NORMAL)
114        if flag != cvode.CV_SUCCESS:
115            break
116        tt.append(tout)
117        y0.append(y[0])
118        y1.append(y[1])
119        y2.append(y[2])
120
121    ax1.plot(tt,y0, color='b')
122    ax2.plot(tt,y1, color='b')
123    ax3.plot(tt,y2, color='b')
124
125
126 def solveEuler():
127    tt = np.linspace(tstart, tend, int((tend-tstart)/h))
128    y = np.zeros([6, len(tt)])
129    u = np.zeros(6)
130
131    y[0,0] = 0.16
132    y[1,0] = 0.055
133    y[2,0] = 0.15
134    y[3,0] = 0
135    y[4,0] = 0
136    y[5,0] = 0
137
138    for i in range(0,len(tt))[:-1]:
139        f(0, y[:,i], u, None)
140        y[:,i+1] = y[:,i] + h*u[:]
141
142    ax1.plot(tt,y[0,:], color='r')
143    ax2.plot(tt,y[1,:], color='r')
144    ax3.plot(tt,y[2,:], color='r')
145
146 if __name__=='__main__':
147    print 'coefficiente stiffness: ' + str(stiffness())
148    hcr = calcoloHEuler()
149    print 'h critico per Eulero esplicito = ', hcr
150
151    for h in [1e-8, hcr, hcr+1e-10, 1e-4]:
152        print '\nh attuale: ', h
153        figure, (ax1,ax2,ax3) = p.subplots(3,1,sharex=True)
154        ax1.set_title('x1 - (b) bdf, (r) euler')
155        ax2.set_title('x2 - (b) bdf, (r) euler')
156        ax3.set_title('x3 - (b) bdf, (r) euler')
```

```
157         ax1.grid(True); ax2.grid(True); ax3.grid(True)
158         ax1.hold(True); ax2.hold(True); ax3.hold(True)
159
160         solveOdeBDF()
161         solveEuler()
162         p.show() # mostra il grafico
```

Bibliografia

[1] V. Comincioli, *Analisi numerica. Metodi, modelli, applicazioni*, McGraw-Hill, 1995

[2] M. L. Corradini, G. Orlando, *Controllo Digitale di Sistemi Dinamici*, FrancoAngeli, 2005

[3] C. D'Angelo, A. Quarteroni, *Matematica Numerica Esercizi, Laboratori e Progetti*, Springer, 2010

[4] K. De Cock, B. De Moor, W. Minten, W. Van Brempt, H. Verrelst, *A tutorial on PID-control*, Katholieke Universiteit Leuven, 1997

[5] E. Hairer, G. Wanner, *Linear multistep method*, Scholarpedia, 5(4):4591., revision #91436, 2010, `http://www.scholarpedia.org/article/Linear_multistep_method`

[6] A. C. Hindmarsh, R. Serban, *User documentation for* CVODE *v.2.5.0*, Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, 2006

[7] A. Iserles, *A First Course in the Numerical Analysis of Differential Equations*, Cambridge University Press, 2008

[8] *PIC18F2525/2620/4525/4620 Data Sheet*, Microchip Technology, 2008

[9] A. Quarteroni, R. Sacco, F. Saleri, *Matematica Numerica*, Springer, 2008

[10] A. Quarteroni, F. Saleri, *Calcolo Scientifico: Esercizi e problemi risolti con MATLAB e Octave*, Springer, 2008

[11] F. Vahid, T. Givargis, *Embedded System Design: A Unified Hardware/Software Introduction*, Wiley, 2002