Università degli Studi di Cagliari
Facoltà di Ingegneria
Corso di Calcolo Numerico 2 , *Prof*. Giuseppe Rodriguez

CALCOLO DEL POLINOMIO DI MIGLIORE APPROSSIMAZIONE

Fattorizzazione QR nella risoluzione di problemi ai minimi quadrati

Fabrizio Serpi 37322

***a.a.* 2010 / 2011**

Sommario

L'approssimazione di funzioni è una tecnica matematica che viene utilizzata in diverse situazioni. Si ricorre ad essa per esempio nei casi in cui la funzione sia conosciuta solo su base campionaria oppure in casi in cui nonostante la sua conoscenza analitica, essa faccia parte di una espressione matematica più complessa che di fatto può essere semplificata da una sua opportuna approssimazione.

In questo scenario, il polinomio di migliore approssimazione è appunto un polinomio approssimante a cui si ricorre soprattutto nei casi in cui la funzione sia conosciuta su base campionaria. A differenza del polinomio interpolante, il polinomio di migliore approssimazione risulta molto più efficace nei casi in cui i campioni conosciuti della funzione siano affetti da errore.

Il calcolo del polinomio di miglior approssimazione, se effettuato in norma-2, porta alla risoluzione di un problema ai minimi quadrati per il quale possono essere utilizzate le fattorizzazioni matriciali, tra cui la fattorizzazione QR. Quello che andremo a vedere in seguito sarà l'applicazione di tale tecnica alla risoluzione di problemi di approssimazione.

Il presente lavoro consta di due parti principali. La prima parte consiste nell'implementazione e nel test di due fattorizzazioni matriciali. La prima è la fattorizzazione di Cholesky, applicabile a matrici quadrate, simmetriche e definite positive mentre la seconda è la fattorizzazione QR che più in generale può essere utilizzata per la fattorizzazione di matrici rettangolari generiche. La seconda parte invece prevede appunto l'applicazione della fattorizzazione QR nella risoluzione di un problema ai minimi quadrati derivante dal calcolo del polinomio di migliore approssimazione.

Indice

PARTE I Fattorizzazioni Matriciali

La fattorizzazione di una matrice è una tecnica matematica che permette di scrivere una matrice data nel prodotto di due o più matrici, ognuna delle quali può avere delle caratteristiche particolari che permettono di semplificare la risoluzione del problema a cui appartiene la matrice fattorizzata stessa. Ovviamente, perché tali scomposizioni siano possibili, la matrice da fattorizzare deve avere particolari requisiti che variano a seconda della fattorizzazione considerata. Non solo, ognuna di queste scomposizioni introduce determinate semplificazioni, per cui se più di una fattorizzazione può essere utilizzata bisogna tener conto anche del risultato che si vuole ottenere per esempio in termini di velocità di risoluzione del problema o in termini di occupazione di memoria dell'algoritmo risolutivo risultante.

Uno dei problemi classici su cui possono essere utilizzate diverse fattorizzazioni è quello che vede la risoluzione di un sistema lineare, al quale possono essere ricondotti molti altri problemi di diversa natura:

$$A\mathbf{x} = \mathbf{b}, \qquad (I.1)$$

dove, in generale:

- $A \in \mathcal{R}^{m\,x\,n}$ matrice del sistema,
- $\mathbf{x} \in \mathcal{R}^{n}$ vettore delle incognite,
- $\mathbf{b} \in \mathcal{R}^{m}$ vettore dei termini noti.

Come vedremo nella seconda parte, la risoluzione di un problema ai minimi quadrati viene proprio ricondotta alla risoluzione di un sistema lineare non omogeneo.

I.1 Fattorizzazione di Cholesky

La fattorizzazione di Cholesky è una particolare fattorizzazione matriciale che può essere effettuata su matrici del tipo:

1. Non singolari,
2. Quadrate,
3. Simmetriche,
4. Definite positive.

Il vantaggio apportato da questa fattorizzazione è quello di scomporre la matrice a cui viene applicata nel prodotto di due matrici triangolari di cui una è la trasposta dell'altra:

$$A = R^T R, \tag{I.2}$$

dove:

- $A \in \mathcal{R}^{n\,x\,n}$ è una matrice quadrata di generiche dimensioni (n x n),
- $R, R^T \in \mathcal{R}^{n\,x\,n}$ sono le due matrici triangolari delle stesse dimensioni di A, di cui R è la triangolare superiore e R^T, essendo la trasposta di R, è la triangolare inferiore.

Per quanto riguarda il suo utilizzo, tale fattorizzazione risulta un utile strumento nella risoluzione di sistemi lineari, permettendo di ottenere, a partire da (1), due sistemi lineari triangolari di più facile risoluzione:

$$\begin{cases} R^T \mathbf{y} = \mathbf{b} \\ R\,\mathbf{x} = \mathbf{y} \end{cases}. \tag{I.3}$$

L'algoritmo che permette il calcolo effettivo dei fattori con cui la matrice A viene scomposta può essere ricavato partendo dall' osservazione della scomposizione stessa. Infatti, essendo i due fattori uno il trasposto dell'altro, possiamo costruire un algoritmo che determini direttamente i singoli elementi che poi andranno a riempire il triangolo superiore di R e il triangolo inferiore di R^T:

$$A = R^T R = \begin{bmatrix} r_{11} & & 0 \\ \vdots & \ddots & \\ r_{1n} & \cdots & r_{nn} \end{bmatrix} \begin{bmatrix} r_{11} & \cdots & r_{1n} \\ & \ddots & \vdots \\ 0 & & r_{nn} \end{bmatrix}. \qquad \text{(I.4)}$$

Esprimiamo gli elementi di A a partire dal prodotto di R e della sua trasposta nella forma:

$$a_{ij} = \sum_{k=1}^{i} r_{ki} r_{kj} \qquad i \leq j$$

Partendo da questa espressione, esplicitando l'ultimo termine della sommatoria e distinguendo i due casi per i < j e i = j, otteniamo le seguenti espressioni per gli elementi di R in funzione di quelli di A:

$$r_{ij} = \frac{1}{r_{ii}} \left(a_{ij} - \sum_{k=1}^{i-1} r_{ki}\, r_{kj} \right) \qquad i < j$$

$$r_{jj} = \left(a_{jj} - \sum_{k=1}^{j-1} r_{kj}^2 \right)^{\frac{1}{2}} \qquad i = j$$

Da notare che la proprietà di matrice definita positiva garantisce, nel calcolo di r_{jj} , un radicando sempre positivo. Per quanto riguarda la complessità computazionale, l'algoritmo di Cholesky necessità di $O\left(\frac{n^3}{6}\right)$ moltiplicazioni.

I.1.2 Implementazione e Test

L'implementazione dell'algoritmo può essere fatta in due modi principali. La matrice R infatti può essere costruita sia per righe che per colonne dove la sua costruzione per colonne ha il vantaggio di poter aggiornare la fattorizzazione già effettuata nel caso in cui alla matrice A vengano aggiunte nuove righe e colonne oltre la *n*-esima. Di seguito ne diamo una implementazione in pseudo-codice.

ALGORITMO I.1 **Fattorizzazione di Cholesky per colonne.**

1. for $j = 1, \ldots, n$
   a. for $i = 1, \ldots, j\text{-}1$
      1. $r_{ij} = \frac{1}{r_{ii}} \left(a_{ij} - \sum_{k=1}^{i-1} r_{ki}\, r_{kj}\right)$
   b. $r_{jj} = \left(a_{jj} - \sum_{k=1}^{j-1} r_{kj}^2\right)^{\frac{1}{2}}$

La stesura vera e propria dell'algoritmo è stata effettuata sul software di simulazione **MatLab**®, il quale fornisce un suo linguaggio di programmazione. La fattorizzazione è stata scritta come una funzione che può essere richiamata da altri programmi che ne avessero necessità (si veda a tal proposito la parola chiave del linguaggio `function`).

File I.1 `myCholesky.m`

```
%Algoritmo di Cholesky per la fattorizzazione di una matrice simmetrica
%definita positiva. L'algoritmo costruisce la matrice R per colonne.
%Input  => A ,   matrice quadrata, simmetrica, definita positiva.
%Output => R,    matrice quadrata triangolare superiore.

function R = myCholesky(A)

    %Verifiche che la matrice sia fattorizzabile secondo Cholesky.
    if(isempty(A) == 1)
        disp( 'La matrice inserita è vuota' )
        R = 1;
        return;
    end

    [m,n] = size(A);

    if (m ~= n)
        disp('La matrice deve essere quadrata')
        R = 1;
        return;
    end
```

```
    if(A' ~= A)
        disp('La matrice deve essere simmetrica')
        return;
    end

    if(det(A) == 0)
        disp('La matrice deve essere non singolare')
        return;
    end

    %Fattorizzazione di Cholesky per colonne.

    Rt  = zeros(n);
    R   = zeros(n);

    for j = 1 : n
        for i = 1 : j-1
           s1  = 0;
           for k = 1 : i-1
               s1 = s1 + (R(k,i) * R(k,j));
           end

           if( i == 1)
              R(i,i) = sqrt(A(1,1));
           end

           R(i,j) = (A(i,j) - s1) / R(i,i);
        end

        s2  = 0;
        for h = 1 : j-1
           s2 = s2 + (R(h,j) * R(h,j));
        end
        R(j,j) = sqrt(A(j,j) - s2);

    end

end
```

Come possiamo vedere dal listato, la prima parte della funzione si occupa di effettuare i controlli sulle proprietà della matrice passata come argomento. Tali controlli sono necessari al fine di determinare se la matrice sia fattorizzabile o meno secondo Cholesky. Per problemi di stabilità numerica (si veda l'Appendice A), si è scelto di non effettuare il controllo per la definita positività. Tale scelta non modifica l'esito delle prove se non in termini di un leggero aumento di velocità nell'esecuzione del codice stesso e le rende comunque possibili in quanto le matrici test utilizzate sono definite positive.

Per quanto riguarda il test del codice, si è scelto di applicare la function scritta alla risoluzione di un sistema lineare la cui soluzione è nota, confrontandone le prestazioni con l'algoritmo di Cholesky e l'algoritmo di Gauss nativi di MatLab. Come termine di paragone è stata scelta la velocità di calcolo della soluzione, in funzione delle dimensioni della matrice del sistema.

File I.2 `myCholesky_vs_Cholesky_Gauss.m`

```
%Confronto tra la fattorizzazione di Cholesky custom, la fattorizzazione
%di Cholesky di Matlab e l'algoritmo di Gauss per la risoluzione di un
%sistema lineare quadrato a matrice simmetrica e definita positiva. Il confronto viene
%effettuato confrontando la velocita d'esecuzione in funzione dell'ordine
```

```
%della matrice del sistema. Nel caso in cui viene utilizzata la
%fattorizzazione myCholesky vengono anche utilizzzate due funzioni apposite
%per la risoluzione di sistemi triangolari (ltss e utss) mentre nel caso in
%cui viene utilizzata la fattorizzazione di Cholesky di MatLab i due
%sistemi triangolari risultanti vengono risolti mediante la funzione \ .

disp('Confronto tra myCholesky e alcuni algoritmi MatLab')
select = input('Selezionare la matrice da utilizzare per il test\nRandom = 1\nHilbert =
2\nSelezionare:  ');
N = input('Scegliere il numero N di prove da effettuare =  ');
n = zeros(1,N);
p = input('Scegliere il passo p di variazione delle dimensioni di B = ');
myt = 0;
tc  = 0;
tg  = 0;

for i = 1 : N
   i
   %Selezione della matrice test
   switch select
       case 1
            A   = rand(i*p);
            B   = A'*A;
       case 2
            B = hilb(i*p);
       otherwise
            disp('La matrice selezionata non esiste')
            return;

   end

   n(i)= i*p;
   b = rand(n(i),1);
   x = zeros(1,n(i));

   %Fattorizzazione di Cholesky
   tic;
   myR  = myCholesky(B);
   myRt = myR';

   %Risoluzione del sistema triangolare inferiore R'*y=b
   y = ltss(myRt,b);

   %Risoluzione del sistema triangolare superiore R*x=y
   x = utss(myR,y);
   myt(i) = toc;

   %Risoluzione del sistema lineare mediante la fattorizzazione
   %di Cholesky nativa di MatLab
   tic;
   R  = chol(B);
   Rt = R';
   u = Rt \ b;
   w = R \ u;
   tc(i) = toc;
   %Risoluzione del sistema lineare mediante l'algoritmo di Gauss
   %nativo di MatLab
   tic;
   z = B \ b;
   tg(i) = toc;

end
%Plot dei risultati
semilogy(n,myt,'r*--',n,tc,'b*--',n,tg,'k*--');
legend('myCholesky','Cholesky','Gauss',2);
title('Confronto tra algoritmi per la risoluzione di sistemi lineari');
xlabel('n [Ordine della Matrice del sistema]');
ylabel('t [Tempo impiegato per la risoluzione]');
```

Appena lanciato, il programma richiede l'inserimento dei parametri N e p. Il primo mi dice quante sono le prove che il programma dovrà effettuare mentre il secondo mi dice di quanto deve variare l'ordine della matrice da una prova alla successiva. Ne consegue che il prodotto N*p mi dà l'ordine massimo che il sistema test può raggiungere.

Per la risoluzione del sistema lineare mediante la function myCholesky sono state implementate altre due functions, ltss e utss (files `ltss.m` e `utss.m`), che rispettivamente si occupano della risoluzione di un sistema lineare triangolare inferiore e di un sistema lineare triangolare superiore.

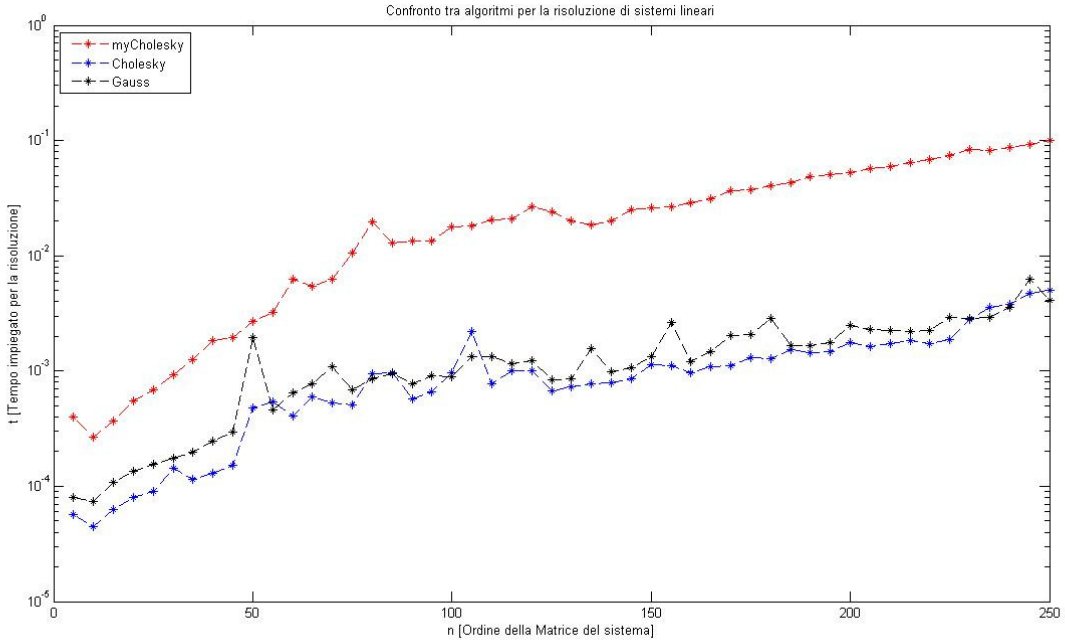

Figura I.1: Confronto in termini di velocità di calcolo nella risoluzione di un sistema lineare test con matrice random.

Per quanto riguarda invece la rilevazione dei tempi d'esecuzione è stata usata la coppia di istruzioni tic e toc, le quali rispettivamente fanno partire e fermano il cronometro. In tal modo viene misurato il tempo d'esecuzione delle istruzioni racchiuse dalla coppia tic-toc. Nella figura I.1 vediamo plottati i risultati di un test consistente in $N = 50$ prove consecutive eseguite su un sistema la cui matrice, di tipo random, varia di dimensione con un passo di $p = 5$ da una prova all'altra. Si vede chiaramente come l'implementazione myCholesky sia comunque più lenta degli algoritmi di MatLab che più o meno hanno la stessa velocità in termini di ordini di grandezza. Vediamo che all'aumentare delle dimensioni del sistema la differenza di velocità aumenta, attestandosi sopra l'ordine di grandezza per $n = 250$.

Analoghe prove sono state effettuate con un sistema test la cui matrice è stata scelta di Hilbert. In questo caso il test è stato limitato nelle dimensioni della matrice dal cattivo condizionamento della stessa. Infatti la funzione chol.m di MatLab effettua la verifica sulla matrice da fattorizzare che questa sia definita positiva o meno.

Tale riconoscimento però fallisce nel momento in cui la matrice supera la dimensione di n = 13 (vedi appendice A) per cui questa è stata scelta come dimensione massima raggiungibile. Il test è stato quindi effettuato con un valore dei parametri di N = 13 e p = 1. I risultati sono rappresentati nella figura I.2. Ancora una volta, l'algoritmo myCholesky risulta più lento rispetto agli algoritmi di MatLab mentre Cholesky di MatLab risulta in assoluto il più veloce, anche rispetto all'algoritmo di Gauss.

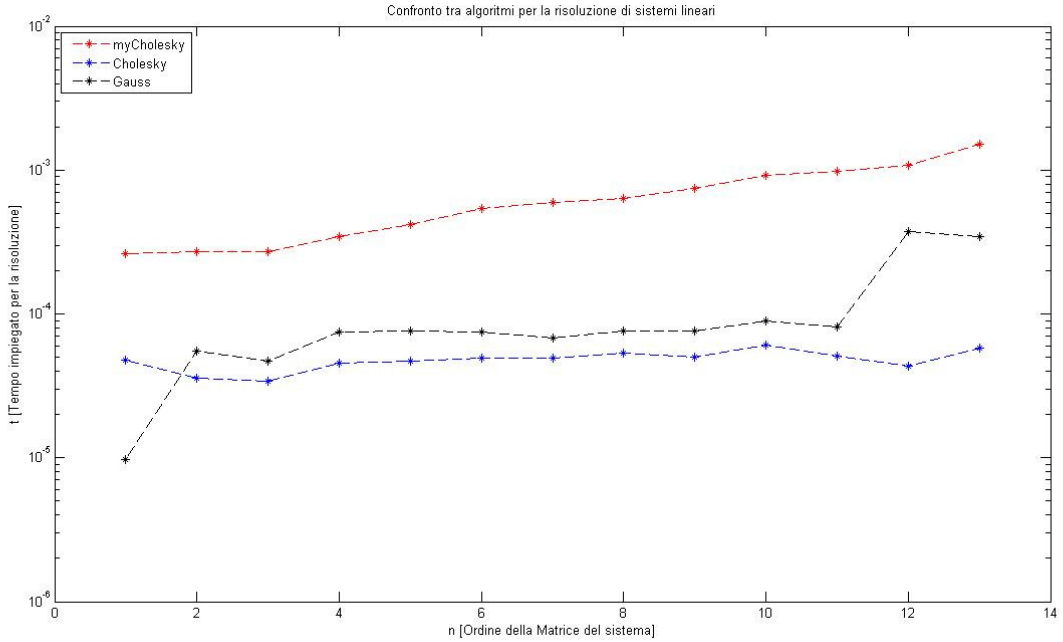

Figura I.2: Confronto in termini di velocità di calcolo nella risoluzione di un sistema lineare test con matrice di Hilbert.

I.2 Fattorizzazione QR

La fattorizzazione QR è una fattorizzazione che può essere applicata ad una matrice rettangolare generica. Data quindi una matrice A tale che A ϵ $\mathcal{R}^{m\,x\,n}$, questa può essere scritta come prodotto di altre due matrici:

$$A = QR \tag{I.5}$$

Dove:

- Q ϵ $\mathcal{R}^{m\,x\,m}$ è una matrice ortogonale, tale che $Q^T Q = QQ^T = I$.
- R ϵ $\mathcal{R}^{m\,x\,n}$ è una matrice triangolare superiore delle stesse dimensioni di A.

Quando A è una matrice quadrata non singolare, la fattorizzazione QR può essere usata, così come abbiamo già visto nel caso di quella di Cholesky, per risolvere sistemi lineari del tipo $Ax = b$. Sostituendo infatti alla matrice A la sua fattorizzazione, otteniamo due sistemi di più facile risoluzione:

$$\begin{cases} Qc = b \\ Rx = c \end{cases} \tag{I.6}$$

Il primo infatti è un sistema ortogonale, di cui conosciamo l'inversa della matrice che è pari alla trasposta, e la cui soluzione risulta quindi $c = Q^T b$. Il secondo invece essendo un sistema triangolare superiore si risolve con l'algoritmo di backward substitution.

Per quanto riguarda la realizzazione pratica della fattorizzazione QR, ci sono vari modi per poterla ottenere. Nel nostro caso abbiamo scelto di realizzarla mediante l'algoritmo che sfrutta le matrici elementari di Householder (vedi Appendice B). L'idea che sta alla base di questo algoritmo è quella di costruire passo dopo passo la matrice triangolare superiore R a partire dalla matrice da fattorizzare A a cui vengono appunto applicate tali matrici elementari. Ad ogni passo dell'algoritmo la matrice di Householder corrente si occuperà quindi di azzerare una parte della colonna che sto considerando in quel particolare passo.

Per capire come questo venga fatto, ricordiamo che una matrice di Householder in questo caso viene costruita in modo tale che questa verifichi la relazione

$$Hx = ke_1 \tag{I.7) (B.2}$$

dove $k \in \boldsymbol{\mathcal{R}}$ è una costante e e_1 è il primo versore della base canonica di $\boldsymbol{\mathcal{R}}^n$. La cosa importante da notare è il secondo membro di tale uguaglianza. Infatti la precedente espressione non dice altro che il prodotto di H per un generico vettore x, trasforma il vettore in modo tale che questo abbia il solo primo elemento non nullo e di valore k e tutti gli altri nulli. Se pensiamo poi di sostituire a x un vettore colonna di A, questo ci fa intuire come le matrici elementari di Householder possono essere usate per la triangolarizzazione di A.

Per i nostri scopi considereremo matrici di dimensioni m x n, dove $m \geq n$. Supponiamo senza perdita di generalità che inizialmente la matrice da fattorizzare A sia quadrata. Nel caso rettangolare dovremo solo aggiungere un ulteriore passo per completarne la fattorizzazione. L'algoritmo che andremo a descrivere genera una successione di matrici $A^{(i)}$, $i = 1, \ \dots \ ,n$ tale che la matrice $A^{(n)}$ sia triangolare superiore, ovvero sia la matrice R cercata. Come si può intuire, la costruzione di R termina in n-1 passi e a quello che dovrebbe essere l'n-esimo passo, ho direttamente il risultato delle trasformazioni. Scriviamo la matrice A in termini dei suoi vettori colonna come

$$A = [\, a_1 \ \ a_2 \ \ a_3 \ \ \dots \ \ a_j \ \ \dots \ \ a_n \,]$$

PASSO 1

Al primo passo dell'algoritmo poniamo

$$A = A^{(1)} = [a_1^{(1)} \ \ a_2^{(1)} \ \ \dots \ \ a_j^{(1)} \ \ \dots \ \ a_n^{(1)}]$$

e costruiamo la matrice di Householder H_1 utilizzando la prima colonna di A (vedi Appendice B)

$$H_1 a_1^{(1)} = k_1 e_1$$

Moltiplichiamo a sinistra la matrice A per la matrice H_1, ottenendo così la matrice

$$A^{(2)} = H_1 A^{(1)} = [a_1^{(2)} \;\; a_2^{(2)} \;\; \dots \;\; a_j^{(2)} \;\; \dots \;\; a_n^{(2)}]$$

con

$$a_1^{(2)} = k_1 e_1, \qquad a_j^{(2)} = H_1 a_j^{(1)}, \qquad j = 2, \;\dots\; n$$

La matrice ottenuta ha la seguente struttura

$$A^{(2)} = \begin{bmatrix} k_1 & a_{12}^{(2)} & \dots & a_{1n}^{(2)} \\ 0 & a_{22}^{(2)} & \dots & a_{2n}^{(2)} \\ \vdots & \vdots & \cdots & \vdots \\ 0 & a_{n2}^{(2)} & \cdots & a_{nn}^{(2)} \end{bmatrix} = \begin{bmatrix} k_1 & v_1^T \\ \mathbf{0} & \hat{A}^{(2)} \end{bmatrix}$$

dove $\mathbf{0}$ è un vettore colonna di zeri di dimensione n-1 e $\hat{A}^{(2)}$ è una sottomatrice quadrata di dimensione anche essa n-1. Tale sottomatrice è proprio quella da cui partiremo nel passo successivo.

PASSO 2

Al secondo passo dell'algoritmo partiamo dalla sottomatrice $\hat{A}^{(2)}$ e ripetiamo quanto fatto al passo 1. Scriviamo

$$\hat{A}^{(2)} = [\hat{a}_2^{(2)} \;\; \hat{a}_3^{(2)} \;\; \dots \;\; \hat{a}_j^{(2)} \;\; \dots \;\; \hat{a}_n^{(2)}]$$

e costruiamo la matrice elementare di Householder $\hat{H}_2$ di dimensione n-1 tale che

$$\hat{H}_2 \hat{a}_2^{(2)} = k_2 e_1 ,$$

dove il vettore e_1 ha in questo caso dimensioni n-1. Prima di applicare $\hat{H}_2$ ad $A^{(2)}$ dobbiamo compiere l'operazione chiamata orlatura, in modo da aumentare le dimensioni di $\hat{H}_2$ per renderle compatibili a quelle della matrice a cui viene moltiplicata. Tale operazione consiste semplicemente nell' aggiungere un numero uguale di righe e colonne della matrice identità di ordine n. Nel caso specifico aggiungeremo ad $\hat{H}_2$ la prima riga e la prima colonna di I, ottenendo

$$H_2 = \begin{bmatrix} 1 & 0 & .. & 0 \\ 0 & & & \\ : & & \hat{H}_2 & \\ 0 & & & \end{bmatrix} = \begin{bmatrix} 1 & \mathbf{0}^T \\ \mathbf{0} & \hat{H}_2 \end{bmatrix}.$$

Moltiplicando infine a sinistra della matrice $A^{(2)}$ otteniamo

$$A^{(3)} = H_2 A^{(2)} = \begin{bmatrix} 1 & \mathbf{0}^T \\ \mathbf{0} & \hat{H}_2 \end{bmatrix} \begin{bmatrix} k_1 & v_1^T \\ \mathbf{0} & \hat{A}^{(2)} \end{bmatrix} = \begin{bmatrix} k_1 & v_1^T \\ \mathbf{0} & \hat{H}_2 \hat{A}^{(2)} \end{bmatrix}$$

ovvero

$$A^{(3)} = \begin{bmatrix} k_1 & * & .. & .. & * \\ 0 & k_2 & * & .. & * \\ : & 0 & & & \\ : & : & & \hat{A}^{(3)} & \\ 0 & 0 & & & \end{bmatrix}.$$

La sottomatrice $\hat{A}^{(3)}$, di dimensione n-2, è la matrice da cui partiremo al passo 3. Osserviamo che sino a qui siamo riusciti ad azzerare la parte desiderata delle prime due colonne.

PASSO i-esimo

Dopo aver analizzato i primi due passi dell'algoritmo, possiamo passare al generico passo, che si snoderà in maniera del tutto analoga a quelli precedenti. La matrice prodotta dall'iterazione precedente sarà della forma

$$A^{(i)} = \begin{bmatrix} k_1 & * & \cdots & \cdots & \cdots & * \\ 0 & \ddots & \ddots & & & \vdots \\ \vdots & \ddots & k_{i-1} & * & \cdots & * \\ \vdots & & 0 & & & \\ \vdots & & \vdots & & \hat{A}^{(i)} & \\ 0 & \cdots & 0 & & & \end{bmatrix} = \begin{bmatrix} A_{11}^{(i)} & A_{12}^{(i)} \\ 0 & \hat{A}^{(i)} \end{bmatrix},$$

dove

$$\hat{A}^{(i)} = [\, \hat{a}_i^{(i)} \;\; \hat{a}_{i+1}^{(i)} \;\; \ldots \;\; \hat{a}_n^{(i)}].$$

La matrice elementare di Householder $\hat{H}_i$ sarà in questo caso costruita in base alla relazione

$$\hat{H}_i \hat{a}_i^{(i)} = k_i e_1.$$

Dopo aver orlato $\hat{H}_i$ con i-1 righe e colonne della matrice identità avrò

$$A^{(i+1)} = H_i A^{(i)} = \begin{bmatrix} I_{i-1} & 0 \\ 0 & \hat{H}_i \end{bmatrix} \begin{bmatrix} A_{11}^{(i)} & A_{12}^{(i)} \\ 0 & \hat{A}^{(i)} \end{bmatrix} = \begin{bmatrix} A_{11}^{(i)} & A_{12}^{(i)} \\ 0 & \hat{H}_i \hat{A}^{(i)} \end{bmatrix},$$

dove la matrice $A^{(i+1)}$ ha la struttura

$$A^{(i+1)} = \begin{bmatrix} k_1 & * & \cdots & \cdots & \cdots & * \\ 0 & \ddots & \ddots & & & \vdots \\ \vdots & \ddots & k_i & * & \cdots & * \\ \vdots & & 0 & & & \\ \vdots & & \vdots & & \hat{A}^{(i+1)} & \\ 0 & \cdots & 0 & & & \end{bmatrix}.$$

Dopo n-1 passi otterrò la matrice triangolare superiore della fattorizzazione

$$A^{(i+1)} = H_{n-1}A^{(n-1)} = \begin{bmatrix} k_1 & * & \cdots & * \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & * \\ 0 & \cdots & 0 & k_n \end{bmatrix} = R$$

Se andiamo ad analizzare quello che viene fatto nei passi dell'algoritmo, vediamo che in sostanza possiamo formulare la triangolarizzazione di A in un'unica espressione inserendo tutte le matrici di Householder utilizzate

$$R = A^{(n)} = H_{n-1}H_{n-2}H_{n-3} \dots H_1A^{(1)} = Q^TA$$

e poiché la matrice

$$Q = H_1H_2 \quad \dots \quad H_{n-1}$$

é ortogonale, in quanto prodotto di matrici ortogonali, la precedente relazione mi permette di scrivere A come

$$A = QR.$$

Che non è altro che la fattorizzazione QR cercata.

Per quanto riguarda il caso di matrice rettangolare A di dimensioni m x n, con $m \geq n$, la fattorizzazione richiede un ulteriore passo rispetto ai precedenti. Il passo n-esimo è necessario per poter azzerare parte dell'ultima colonna, ed esattamente la parte che va dalla riga n+1 alla riga m. Dopo tale passo otterrò la matrice

$$R = A^{(n+1)} = \begin{bmatrix} k_1 & * & \cdots & * \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & * \\ \vdots & & \ddots & k_n \\ \vdots & & & 0 \\ \vdots & & & \vdots \\ 0 & \cdots & \cdots & 0 \end{bmatrix} = \begin{bmatrix} R_1 \\ 0 \end{bmatrix},$$

dove R_1 è una matrice n x n triangolare superiore, non singolare se la matrice A è a rango pieno. Analogamente al caso di matrice quadrata, ottengo

$$R = H_nH_{n-1}H_{n-2}H_{n-3} \dots H_1A^{(1)} = Q^TA$$

da cui

$$Q = H_1 H_2 \ \dots \ H_{n-1} H_n$$

$$A = QR$$

Per quanto riguarda la complessità computazionale l'algoritmo richiede $O\left(\frac{2}{3}n^3\right)$ moltiplicazioni, mentre nel caso di matrice rettangolare sono richieste $O\left(n^2\left(m - \frac{1}{3}n\right)\right)$ moltiplicazioni.

I.2.2 Implementazione e Test

L'implementazione dell'algoritmo segue fedelmente quanto visto nel paragrafo precedente. Per quanto riguarda la costruzione della matrice di Householder è stato implementato l'algoritmo B.2 visto nell'Appendice B. La fattorizzazione QR è stata implementata come una funzione da poter richiamare all'interno di altri script che ne avessero bisogno così come era stato fatto per la fattorizzazione di Cholesky. Osservando lo script riportato possiamo facilmente notare il codice che implementa le varie parti della fattorizzazione già viste.

File I.3 `myQR.m`

```
%Algoritmo per la fattorizzazione QR di una matrice di dimensioni m x n.
%L'algoritmo è implementato tramite la fattorizzazione di Householder.

%INPUT  A     matrice da fattorizzare.
%OUTPUT Q,R   fattori della matrice A

function [Q,R]= myQR(A)

    %Verifiche sulla matrice da fattorizzare.

    if(isempty(A) == 1)
        disp( 'La matrice inserita è vuota' )
        Q = 1;
        R = 1;
        return;
    end

    [m,n] = size(A);

    if(m == n)
        disp( 'La matrice da fattorizzare è quadrata' )
    else
        disp( 'La matrice da fattorizzare è rettangolare' )
    end

    N = min([m,n]);
    M = 0;
```

```
    if(m > n)
        M = N;
    else
        M = N - 1;
    end

    Q = eye(m);
    R = A;
    b = 0;
    s = 0;
    k = 0;

    %Fattorizzazione QR per una matrice A mxn con m<=n
    for i = 1 : M

        %Matrice di Householder dell'i-esima iterazione
        Ii = eye(m - i + 1);
        s = norm(R(i:m,i));
        b = s*(s + abs(R(i,i)));
        k = -sign(R(i,i))*s;
        vi = R(i:m,i) - k*Ii(:,1);
        Hi= Ii - (vi*vi') / b;

        %Orlatura della matrice di Householder
        H = eye(m);
        H(i:m,i:m) = Hi;

        %Matrici della fattorizzazione
        Q = Q*H;
        R = H*R;

    end

end
```

La function così scritta è stata usata per un test di velocità d'esecuzione in maniera del tutto analoga a quanto si è fatto nel caso di Cholesky. La fattorizzazione QR è stata confrontata con la QR e l'algoritmo di Gauss nativi di MatLab nella risoluzione di sistemi lineari test. La prova consiste nella risoluzione di diversi sistemi lineari a dimensione via via crescente e nel misurare la velocità d'esecuzione del codice in funzione di questa. Anche in questo caso sono state usate le due funzioni tic/toc per rilevare il tempo.

File I.4 `myQR_vs_QR_Gauss.m`

```
%Confronto tra la fattorizzazione QR custom, la fattorizzazione
%QR di Matlab e l'algoritmo di Gauss per la risoluzione di un sistema
%lineare quadrato. Il confronto viene effettuato confrontando la velocita
%d'esecuzione in funzione dell'ordine della matrice del sistema. Nel caso
%in cui viene utilizzata la fattorizzazione myQRgen viene anche utilizzzata
%una funzione apposita per la risoluzione del sistema triangolare (utss)
%mentre nel caso in cui viene utilizzata la fattorizzazione QR di MatLab il
%sistema triangolare viene risolto mediante la funzione \ .

disp('Confronto tra myQR e alcuni algoritmi MatLab')
select = input('Selezionare la matrice da utilizzare per il test\nRandom = 1\nHilbert =
2\nSelezionare:  ');
N = input('Scegliere il numero N di prove da effettuare =  ');
n = zeros(1,N);
p = input('Scegliere il passo p di variazione delle dimensioni di A = ');
```

```
myt = 0;
tq  = 0;
tg  = 0;

for i = 1 : N
   i
   %Selezione della matrice test
   switch select
       case 1
            A   = rand(i*p);
       case 2
            A = hilb(i*p);
       otherwise
            disp('La matrice selezionata non esiste')
            return;
   end

   n(i)= i*p;
   b = rand(n(i),1);
   x = zeros(1,n(i));

   %Fattorizzazione QR di Householder
   tic;
   [myQ, myR] = myQR(A);

   %Risoluzione del sistema ortogonale Q*y=b
   y = myQ'*b;

   %Risoluzione del sistema triangolare superiore R*x=y
   x = utss(myR,y);
   myt(i) = toc;

   %Risoluzione del sistema lineare mediante fattorizzazione QR di MatLab
   tic;
   [Q,R] = qr(A);
   u = Q'*b;
   w = R \ u;
   tq(i) = toc;
   %Fattorizzazione di Gauss nativa di MatLab
   tic;
   z = A \ b;
   tg(i) = toc;

end
semilogy(n,myt,'r*--',n,tq,'b*--',n,tg,'k*--');
legend('myQR','QR','Gauss',2);
title('Confronto tra algoritmi per la risoluzione di sistemi lineari');
xlabel('n [Ordine della Matrice del sistema]');
ylabel('t [Tempo impiegato per la risoluzione]');
```

Quando lo script viene lanciato, viene richiesto l'inserimento di alcuni parametri che servono per dimensionare il test. Come prima cosa viene richiesto quale tipo di matrice da utilizzare per le prove, se random o di Hilbert (vedi appendice A). Gli altri due parametri da inserire sono N, ovvero il numero di prove da effettuare e p, il passo di variazione delle dimensioni della matrice del sistema. Anche in questo caso, insieme a myQR, è stata usata la funzione `utss` per la risoluzione del sistema triangolare superiore risultante dopo la fattorizzazione.

Il test sulle matrici random è stato condotto con un valore dei parametri di N = 50 e p = 5, con una dimensione massima risultante della matrice del sistema di n = N*p = 250. La figura I.3 mostra il confronto tra i tre algoritmi in termini di velocità in funzione dell'ordine della matrice. Si vede chiaramente che gli algoritmi di MatLab, che più o meno si equivalgono,

sono comunque più veloci dell'algoritmo custom utilizzato, differenza che si può stimare in almeno due ordini di grandezza in corrispondenza di una matrice di dimensioni n = 250.

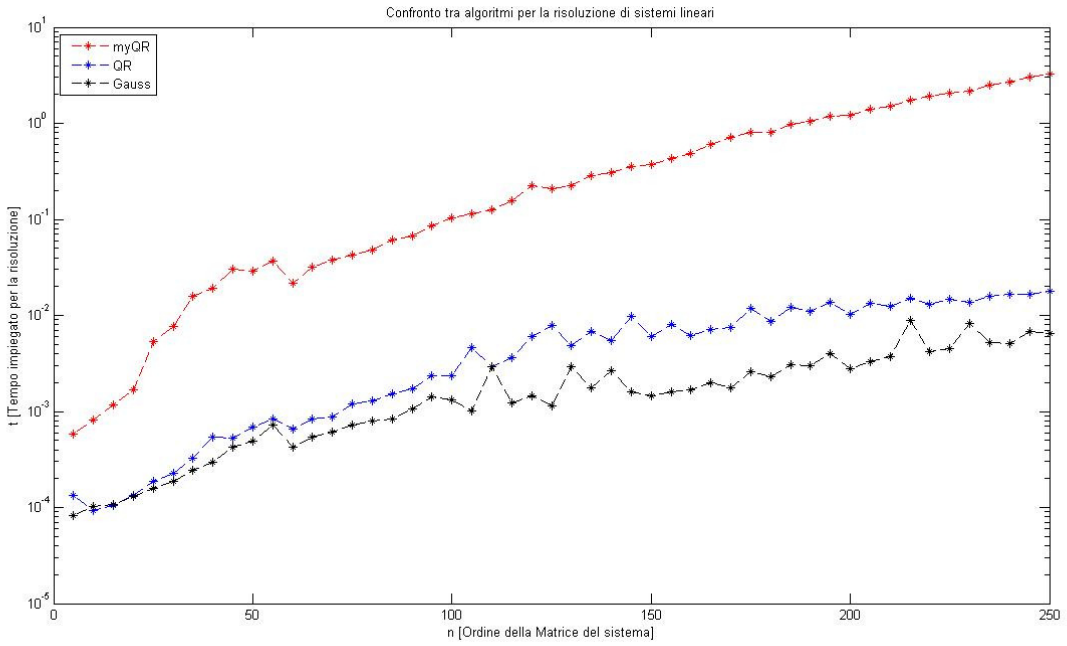

Figura I.3: Confronto in termini di velocità di calcolo nella risoluzione di un sistema lineare test con matrice random.

Il test sulle matrici di Hilbert ha dato risultati analoghi. Condotto anche esso con i valori di parametri N = 50 e p = 5, mostra che gli algoritmi implementati da MatLab hanno la meglio sull'algoritmo QR implementato da noi. La figura I.4 mostra appunto questi risultati.

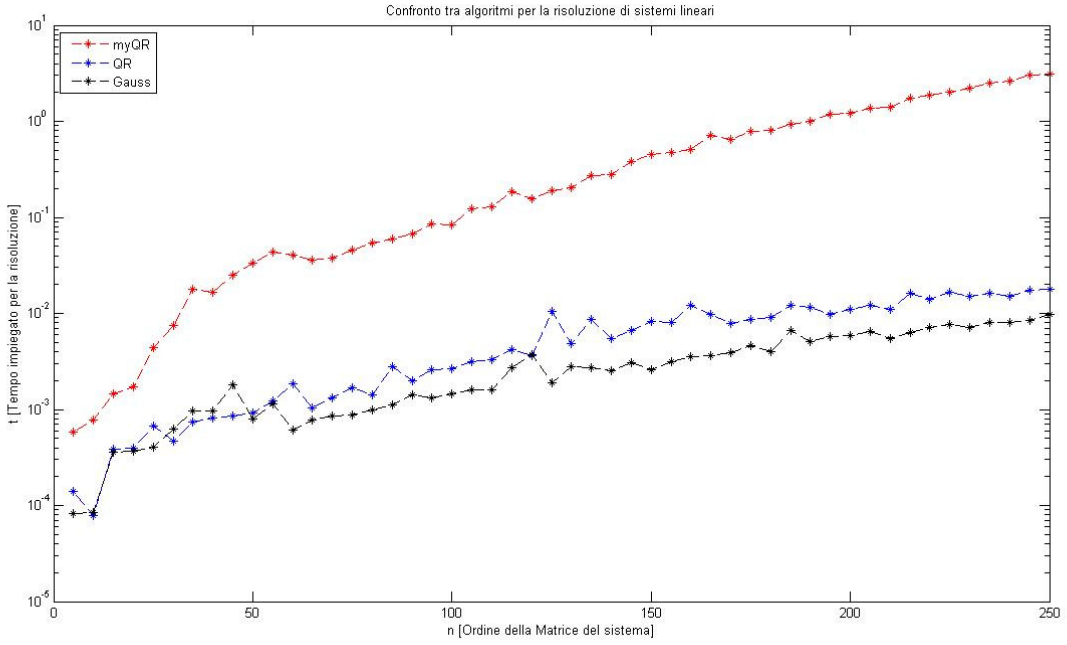

Figura I.4: Confronto in termini di velocità di calcolo nella risoluzione di un sistema lineare test con matrice di Hilbert.

I.3 Test Incrociati

In questa sezione proporremo una serie di test incrociati per verificare l'affidabilità degli algoritmi implementati rispetto alla soluzione calcolata. Andremo, in sostanza, ad effettuare dei test su sistemi lineari a soluzione nota calcolando l'errore commesso dai diversi algoritmi nella risoluzione di questi, al crescere delle dimensioni della matrice del sistema. Tali prove verranno fatte ancora su due tipi di matrici test che sono quella random e quella di Hilbert.

La prima prova vede confrontarsi l'algoritmo myCholesky con l'algoritmo di Gauss nativo di MatLab. Lo script, una volta lanciato, richiede alcune scelte da parte dell'utente. Inizialmente bisogna scegliere quale tipo di matrice utilizzare e poi inserire i soliti due parametri che sono il numero di prove e il passo di crescita della matrice. Il test sulle matrici random è stato effettuato con $N = 50$ prove e con un passo $p = 4$.

File I.5 test_errore_myCholesky.m

```
%Test sull'errore nella soluzione di un sistema lineare mediante
%la fattorizzazione di Cholesky. Per la prova vengono usate una
%matrice di Hilbert e una matrice random.

disp('Confronto tra myCholesky e Gauss')
select = input('Selezionare la matrice da utilizzare per il test\nRandom = 1\nHilbert =
2\nSelezionare:  ');
N = input('Scegliere il numero N di prove da effettuare =  ');
n = zeros(1,N);
p = input('Scegliere il passo p di variazione delle dimensioni di A = ');
error1 = zeros(1,N);
error2 = zeros(1,N);
t   = 0;
myt = 0;

for i = 1 : N
   i
   %Selezione della matrice test
   switch select
       case 1
            B   = rand(i*p);
            A   = B'*B;
       case 2
            A = hilb(i*p);
       otherwise
            disp('La matrice selezionata non esiste')
            return;

   end

   n(i)= i*p;
   x = zeros(1,n(i));
   sol = ones(1,n(i));
   b = A*sol';
   %Fattorizzazione di Cholesky
   tic;
   myR = myCholesky(A);
   myRt = myR';

   %Risoluzione del sistema triangolare inferiore R'*y=b
   y = ltss(myRt,b);

   %Risoluzione del sistema triangolare superiore R*x=y
   x = utss(myR,y);
   myt(i) = toc;
```

```
    %Fattorizzazione di Gauss nativa di MatLab
    tic;
    z = A \ b;
    t(i) = toc;

    er1 = sol - x;
    er2 = sol - z';
    error1(i) = norm(er1);
    error2(i) = norm(er2);
end

semilogy(n,error1,'r*--',n,error2,'k*--');
legend('myCholesky','Gauss',2);
title('Errore nella soluzione del sistema lineare test');
xlabel('n [Ordine della Matrice del sistema]');
ylabel('error [Errore nella soluzione]');
```

Il codice è abbastanza simile a quello già visto precedentemente. Quello che cambia stavolta è che il vettore dei termini noti. Esso infatti è ricavato a partire dalla soluzione che viene fissata inizialmente ed è semplicemente un vettore di elementi tutti pari ad 1. Questo procedimento mi permette poi di calcolare l'errore nella soluzione semplicemente come la norma-2 della differenza tra la soluzione fissata e quella trovata.

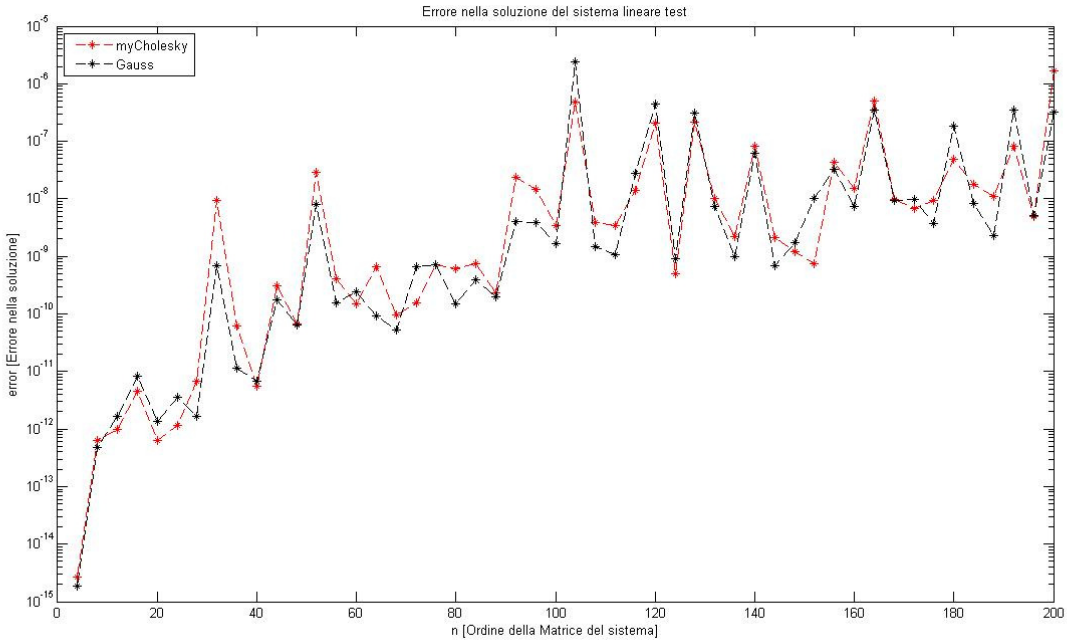

Figura I.5: Confronto tra myCholesky e Gauss in termini di errore di calcolo nella risoluzione di un sistema lineare test con matrice random.

In figura I.5 vediamo plottati i risultati del confronto. In definitiva, possiamo notare che sia myCholesky che Gauss hanno un errore nella risoluzione che è del tutto confrontabile, arrivando a differire in alcune situazioni di al massimo un ordine di grandezza, come per $n = 100$. Per quanto riguarda le matrici test di Hilbert, abbiamo effettuato la prova utilizzando come parametri $N = 28$ e $p = 1$. Inizialmente, non ci è stato possibile aumentare il numero delle prove perché il cattivo condizionamento della matrice ha fatto si che questa venisse riconosciuta come singolare bloccando l'algoritmo.

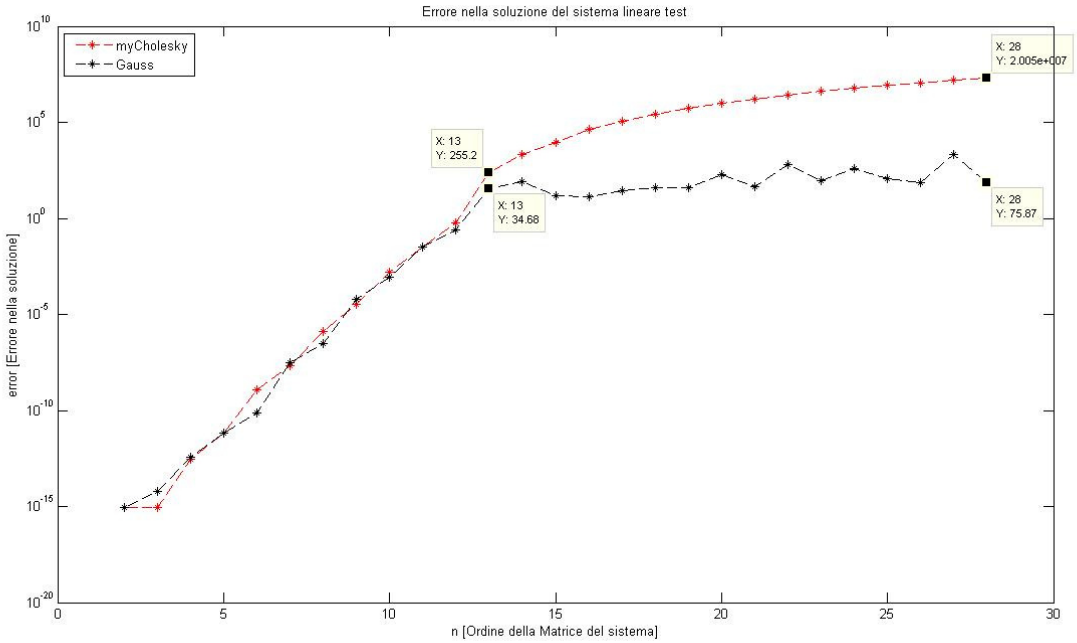

Figura I.6: Confronto tra myCholesky e Gauss in termini di errore di calcolo nella risoluzione di un sistema lineare test con matrice di Hilbert.

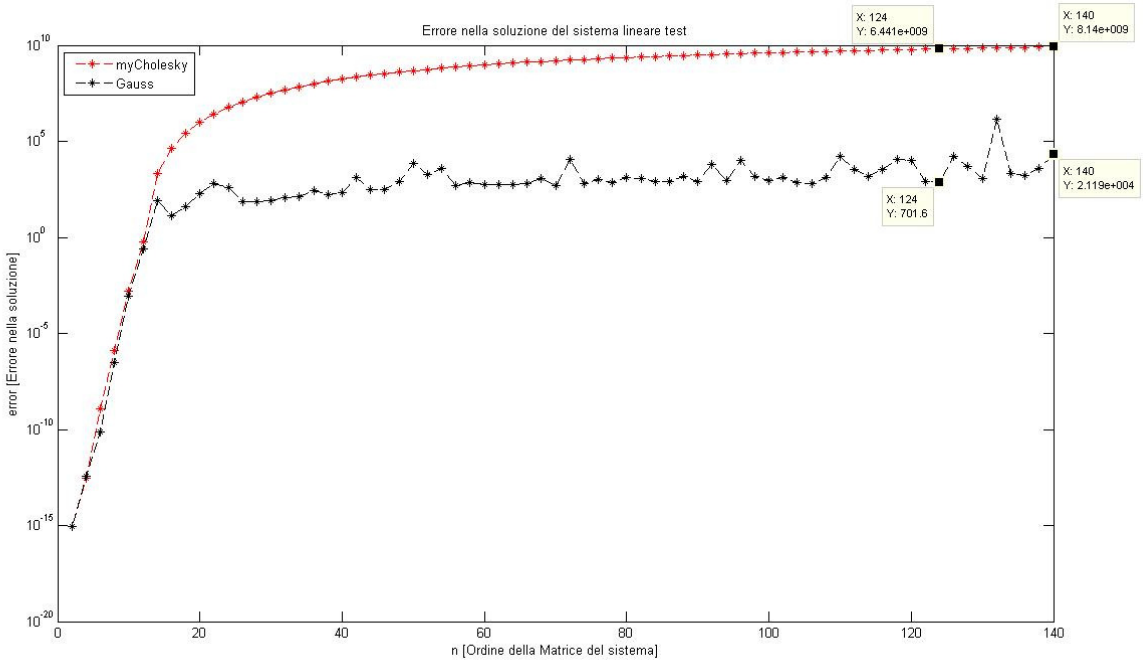

Figura I.7: Confronto tra myCholesky e Gauss in termini di errore di calcolo nella risoluzione di un sistema lineare test con matrice di Hilbert (senza il controllo per la singolarità).

In figura I.6 sono riportati i risultati del test. Vediamo chiaramente che già per un ordine della matrice di $n = 13$, il condizionamento fa si che l'errore aumenti rispetto a quello mostrato dall'algoritmo di Gauss, diventando, per una matrice di dimensioni pari a 28, almeno 5 ordini di grandezza superiore. Per poter utilizzare matrici di dimensione maggiore andiamo a

commentare il codice che controlla nella function myCholesky, la singolarità della matrice passatagli. Questo ci ha permesso di effettuare un ulteriore test in cui abbiamo utilizzato i

parametri N = 70 e p = 2. La figura I.7 riporta i rispettivi risultati. Vediamo che l'errore sulla soluzione ha un andamento tale per i due algoritmi che la differenza tra i due si attesta sui 5 – 6 ordini di grandezza, vedendo comunque myCholesky come algoritmo meno performante.

La seconda prova vede il confronto tra l'algoritmo myQR e l'algoritmo di Gauss nelle stesse condizioni della prova precedente, ovvero nella risoluzione di sistemi lineari test di cui si conosce la soluzione.

File I.6 test_errore_myQR.m

```
%Test sull'errore nella soluzione di un sistema lineare mediante
%la fattorizzazione QR di Householder. Per la prova vengono usate
%una matrice di Hilbert e una matrice random.

disp('Confronto tra myQR e Gauss')
select = input('Selezionare la matrice da utilizzare per il test\nRandom = 1\nHilbert =
2\nSelezionare:  ');
N = input('Scegliere il numero N di prove da effettuare =  ');
n = zeros(1,N);
p = input('Scegliere il passo p di variazione delle dimensioni di A = ');
h = p;
l = p;
n = zeros(1,N);
m = zeros(1,N);
myt = 0;
t   = 0;

error1 = zeros(1,N);
error2 = zeros(1,N);

for i = 1 : N
   i
   %Selezione della matrice test
   switch select
       case 1
            A = rand(i*h,i*l);
       case 2
            A = hilb(i*p);
       otherwise
            disp('La matrice selezionata non esiste')
            return;

   end


   m(i)= i*h;
   n(i)= i*l;

   x = zeros(1,n(i));
   sol = ones(1,n(i));
   b = A*sol';

   %Fattorizzazione QR di Householder
   tic;
   [myQ, myR] = myQR(A);

   %Risoluzione del sistema ortogonale Q*y=b
   y = myQ'*b;

   %Risoluzione del sistema triangolare superiore R*x=y
   x = utss(myR,y);
   myt(i) = toc;
```

```
    %Fattorizzazione di Gauss nativa di MatLab
    tic;
    z = A \ b;
    t(i) = toc;

    er1 = sol - x;
    er2 = sol - z';
    error1(i) = norm(er1);
    error2(i) = norm(er2);
end

semilogy(n,error1,'r*--',n,error2,'k*--');
legend('myQR','Gauss',2);
title('Errore nella soluzione del sistema lineare test');
xlabel('n [Ordine della Matrice del sistema]');
ylabel('error [Errore nella soluzione]');
```

In figura I.8 sono riportati i risultati di un test di parametri N = 75 e p = 4 con matrice random. Si può osservare che entrambi gli algoritmi hanno prestazioni altalenanti, con un errore che può arrivare a differire anche di 3 ordini di grandezza in corrispondenza di una dimensione della matrice pari a 252. L'algoritmo QR risulta essere comunque quello tra i due con le prestazioni in media più scarse.

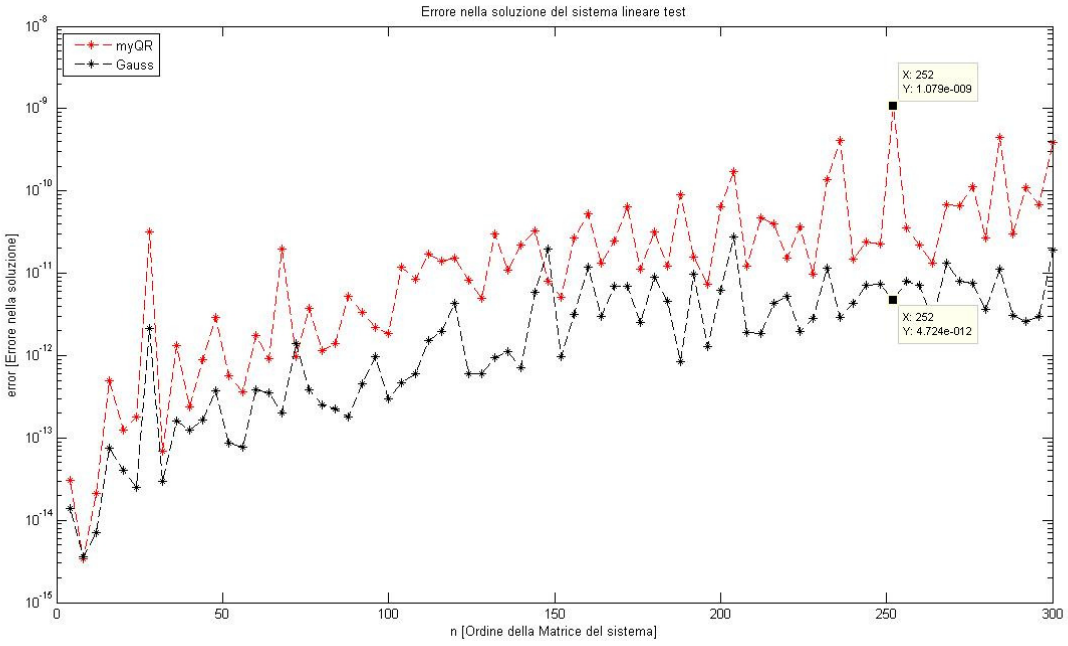

Figura I.8: Confronto tra myQR e Gauss in termini di errore di calcolo nella risoluzione di un sistema lineare test con matrice random.

Per quanto riguarda il test con matrici di Hilbert, le prestazioni saranno sicuramente molto peggiori. Utilizzando gli stessi valori del caso precedente per i parametri N e p otteniamo valori degli errori che già per bassissimi ordini della matrice, sono del tutto paragonabili agli errori commessi nel caso random per elevate dimensioni del sistema. Osservando la figura I.9 infatti, si vede chiaramente che già per una matrice di dimensione n = 12, l'errore è dell'ordine di 10^{-1}, là dove per una dimensione pari a n = 4 era dell'ordine di 10^{-13}. Questo significa che con una variazione nelle dimensioni di 8, ho avuto una variazione dell'errore nella soluzione di ben 12 ordini di grandezza.

Per n che arriva a 300 si vede che l'errore tende ad un ordine di 10^5, valore ben lontano dalle prestazioni degli algoritmi con matrici random che, per tale dimensione, si attestano su un errore di circa 10^{-10} (tra 10^{-9} e 10^{-12}), ben quindici ordini di grandezza inferiore.

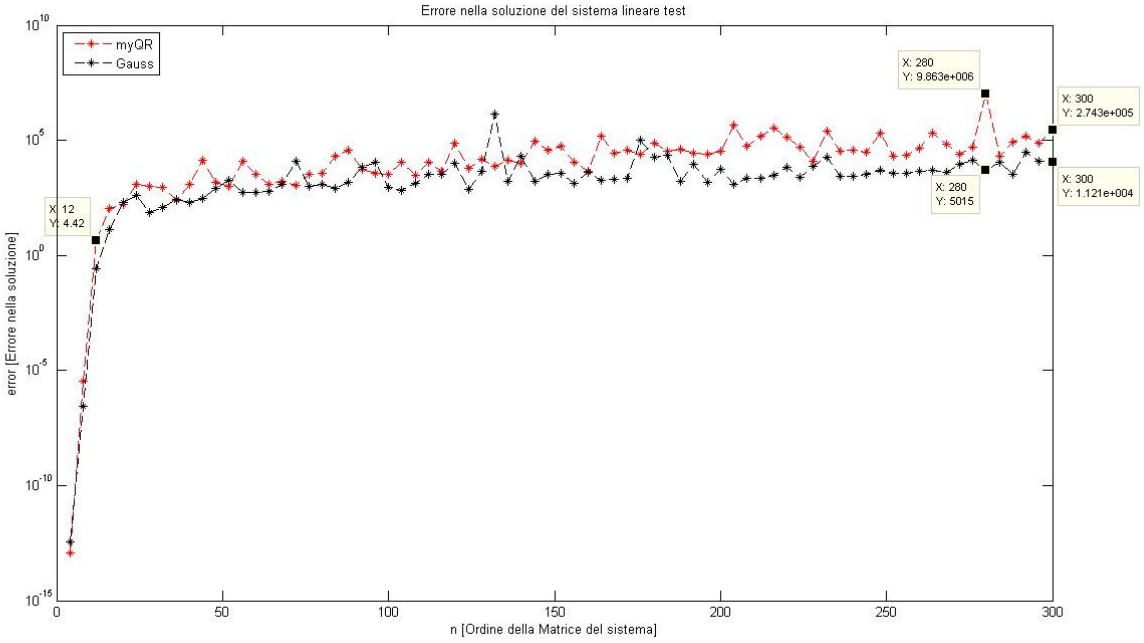

Figura I.9: Confronto tra myQR e Gauss in termini di errore di calcolo nella risoluzione di un sistema lineare test con matrice di Hilbert.

L'ultimo test consiste nel mettere a confronto i due algoritmi implementati, cioè myCholesky e myQR. La comparazione è sempre effettuata su sistemi test la cui soluzione è nota e vede la valutazione sia dei tempi di calcolo, sia dell'errore nella soluzione calcolata, tutto in funzione delle dimensioni della matrice. Lo script, del tutto simile ai precedenti, viene riportato per completezza.

File I.7 myQR vs myCholesky.m

```
%Confronto tra la fattorizzazione di Cholesky e la fattorizzazione QR
%per la risoluzione di sistemi lineari a matrice quadrata, simmetrica,
%definita positiva.

disp('Confronto tra gli algoritmi myCholesky e myQR')
select = input('Selezionare la matrice da utilizzare per il test\nRandom = 1\nHilbert =
2\nSelezionare:  ');
N = input('Scegliere il numero N di prove da effettuare =  ');
n = zeros(1,N);
p = input('Scegliere il passo h di variazione delle dimensioni di B = ');
error1 = zeros(1,N);
error2 = zeros(1,N);
tc  = 0;
tqr = 0;

for i = 1 : N
   i
   %Selezione della matrice test
   switch select
       case 1
            A   = rand(i*p);
            B   = A'*A;
```

```
        case 2
             B = hilb(i*p);
        otherwise
             disp('La matrice selezionata non esiste')
             return;
    end

    n(i)= i*p;
    sol = ones(1,n(i));
    b = B*sol';
    x = zeros(1,n(i));
    z = zeros(1,n(i));

    %Fattorizzazione di Cholesky
    tic;
    myR = myCholesky(B);
    myRt = myR';

    %Risoluzione del sistema triangolare inferiore R'*y=b
    y = ltss(myRt,b);

    %Risoluzione del sistema triangolare superiore R*x=y
    x = utss(myR,y);
    tc(i) = toc;

    %Fattorizzazione QR di Householder
    tic;
    [myQ, myR] = myQR(B);

    %Risoluzione del sistema ortogonale Q*w=b
    w = myQ'*b;

    %Risoluzione del sistema triangolare superiore R*z=w
    z = utss(myR,w);
    tqr(i) = toc;

    er1 = sol - x;
    er2 = sol - z;
    error1(i) = norm(er1);
    error2(i) = norm(er2);

end

%Plot del tempo di calcolo in funzione di n.
subplot(2,1,1),
semilogy(n,error1,'ro--',n,error2,'k^--');
legend('myCholesky','myQR',2);
title('Errore nella soluzione del sistema lineare test');
xlabel('n [Ordine della Matrice del sistema]');
ylabel('error [Errore nella soluzione]');

%Plot degli errori nella soluzione in funzione di n.
subplot(2,1,2),
semilogy(n,tc,'ro--',n,tqr,'k^--');
legend('myCholesky','myQR',2);
title('Tempo impiegato per la risoluzione del sistema test');
xlabel('n [Ordine della Matrice del sistema]');
ylabel('t [Tempo impiegato per la risoluzione]');
```

Il test sulle matrici random mostra che l'algoritmo myCholesky ha prestazioni superiori rispetto all'algoritmo myQR, sia in termini di velocità sia in termini di errore nella soluzione. La figura I.10 mostra i risultati del test effettuato con valori dei parametri di $N = 50$ e $p = 5$. Per quanto riguarda la velocità di calcolo, myCholesky risulta essere sempre superiore rispetto a myQR con una differenza che tende ad aumentare all'aumentare delle dimensioni del sistema e che si attesta sull'ordine di grandezza per una matrice di dimensioni $n = 250$.

Per quanto riguarda l'errore sulla soluzione, i due algoritmi mostrano prestazioni altalenanti, nonostante myCholesky si dimostri in media più performante. Notiamo delle differenze sostanziali in alcuni punti come per esempio per sistemi di dimensioni n = 215, dove si ha una differenza di ben 2 ordini di grandezza, anche se in media la differenza è intorno ad 1 ordine.

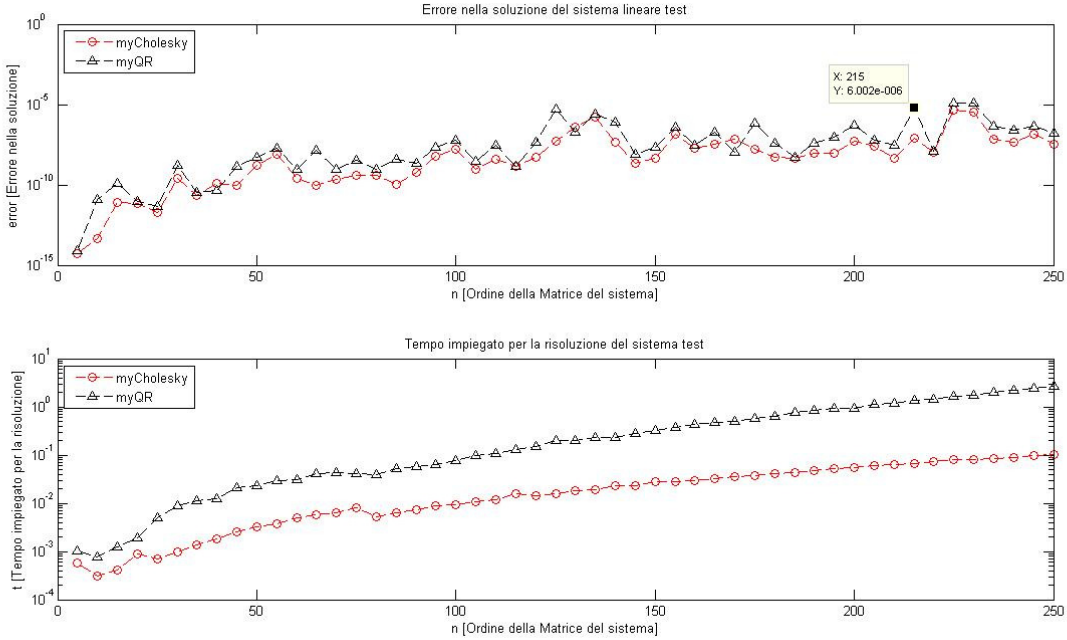

Figura I.10: Confronto tra myQR e myCholesky sia in termini di errore, sia in termini di tempo di calcolo nella risoluzione di un sistema lineare test con matrice random.

Per quanto riguarda il test sulle matrici di Hilbert abbiamo avuto lo stesso problema riscontrato nelle prove precedenti. A causa del'alto condizionamento infatti non si è potuto superare la dimensione di 28 per il sistema test senza incappare in problemi nella verifica della singolarità della matrice. In figura I.11 sono riportati i risultati del test condotto con parametri N = 28 e p = 1. Dai grafici possiamo vedere che, mentre la velocità di calcolo è del tutto simile per i due algoritmi, l'errore commesso nella soluzione è di gran lunga superiore per l'algoritmo myCholesky. Infatti la differenza risulta essere di 4 ordini di grandezza per un sistema di dimensioni n = 28. Eliminando il controllo per la singolarità da myCholesky abbiamo potuto effettuare un ulteriore test, stavolta di parametri N = 62 e p = 4. Dalla figura I.12 vediamo che il trend riscontrato precedentemente prosegue. Infatti la velocità è ancora del tutto comparabile, con myQR leggermente più veloce, anche se gli ordini di grandezza sono i medesimi. L'errore continua ad aumentare all'aumentare delle dimensioni, con myQR che presenta prestazioni nettamente superiori. Infatti per dimensioni n = 216 l'algoritmo myQR presenta un errore di ben 5 ordini di grandezza inferiore. Per n > 216 myCholesky non permette più neanche il calcolo della soluzione del sistema, che non risulta essere più un numero di macchina. La curva dell'errore infatti si interrompe proprio in corrispondenza a tale valore.

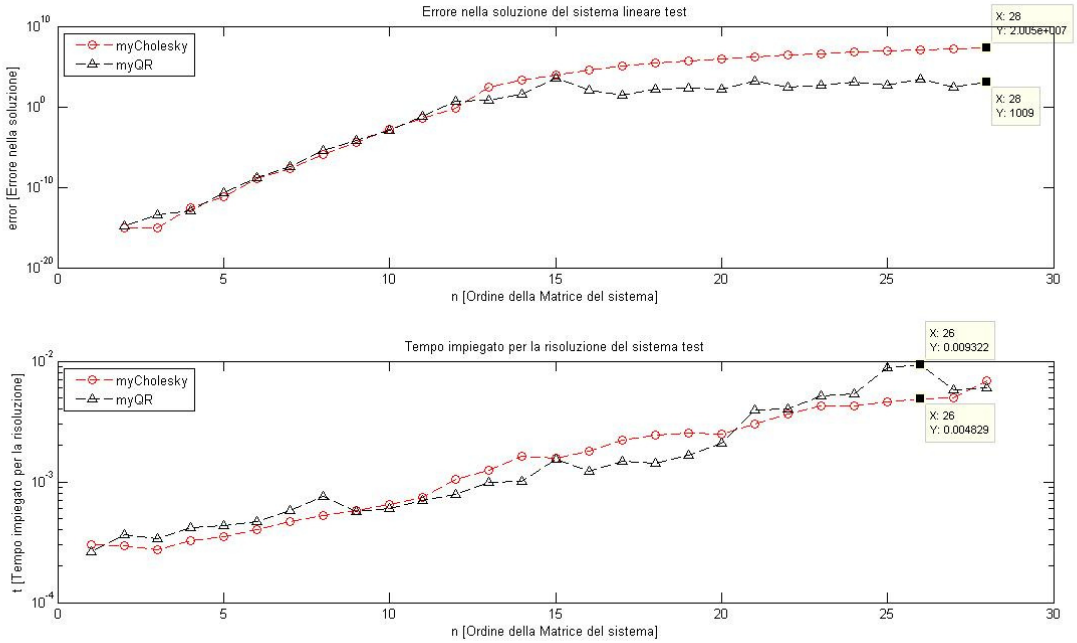

Figura I.11: Confronto tra myQR e myCholesky sia in termini di errore, sia in termini di tempo di calcolo nella risoluzione di un sistema lineare test con matrice di Hilbert.

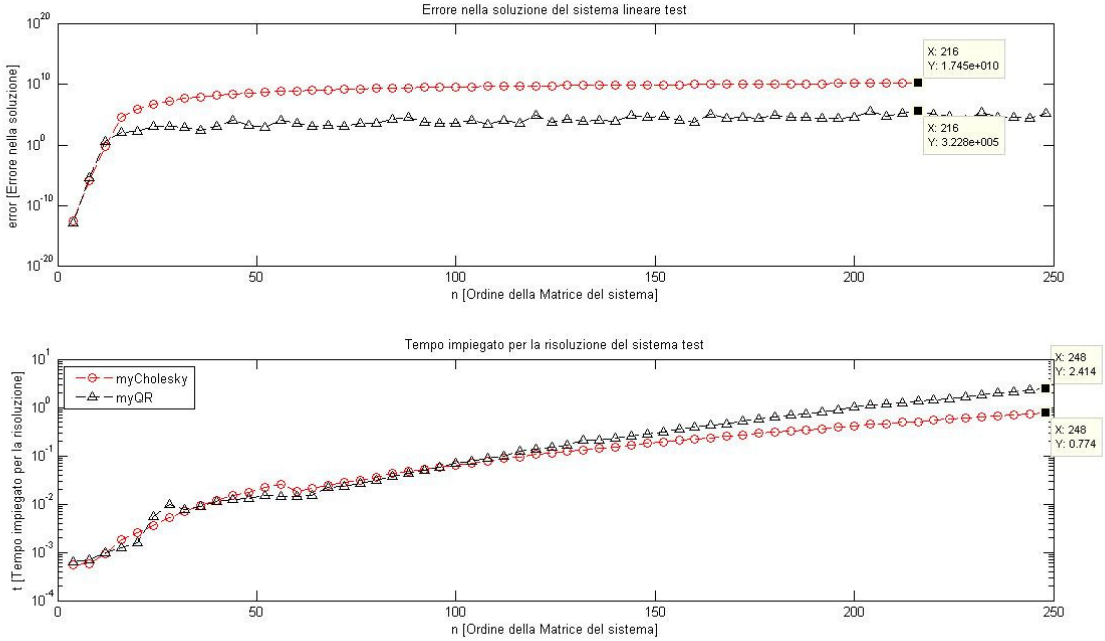

Figura I.12: Confronto tra myQR e myCholesky sia in termini di errore, sia in termini di tempo di calcolo nella risoluzione di un sistema lineare test con matrice di Hilbert (senza controllo singolarità).

PARTE II Polinomio di migliore Approssimazione

II.1 Polinomio Approssimante e Problema ai Minimi Quadrati

Il polinomio di migliore approssimazione, per una data funzione f, è quel polinomio di grado n tale che una norma dell'errore di approssimazione sia minima. In formule possiamo scrivere che, data una funzione f ϵ $L^2_{[a,b]}$ da approssimare, il polinomio di migliore approssimazione p_n è tale per cui

$$\min_{p_n \epsilon\, \Pi_n} \|p_n - f\|. \tag{II.1}$$

Il problema ai minimi quadrati discende dall'espressione precedente nel momento in cui viene scelta la norma-2. Per tale norma infatti, nello spazio $L^2_{[a,b]}$, il problema precedente prende la forma

$$\min_{p_n \epsilon\, \Pi_n} \left(\int_a^b [p_n(x) - f(x)]^2 \right)^{\frac{1}{2}}. \tag{II.2}$$

Il polinomio calcolato mediante la (II.2) prende il nome di **Polinomio di Miglior approssimazione nel senso dei Minimi Quadrati**. Vista l'espressione della norma, e tendo conto del fatto che minimizzare una norma o minimizzarne il suo quadrato è del tutto equivalente, il problema di minimizzazione può così essere espresso nella forma

$$\min_{p_n \epsilon\, \Pi_n} \|p_n - f\|_2^2 . \tag{II.3}$$

Non sempre però la funzione da approssimare è conosciuta su base analitica ma, molto spesso, è conosciuta solo su base campionaria perché per esempio è il risultato di misure sperimentali. In tal caso la (II.2) viene modificata con una discretizzazione della norma. Supponiamo quindi di avere un insieme di m+1 punti di ascisse $\{x_0, x_1, \ \dots \ , x_m\}$ in corrispondenza delle quali sono noti i valori della f(x) $\{y_0, y_1, \ \dots \ , y_m\}$ $(f(x_i) = y_i)$. Dato un polinomio di grado n, per ogni polinomio $p_n \epsilon\, \Pi_n$, con n $\leq$ m, potrò scrivere la norma precedentemente vista nella forma

$$\|p_n - f\| = \left(\sum_{i=0}^{m} [p_n(x_i) - y_i]^2\right)^{\frac{1}{2}}. \tag{II.4}$$

Nel caso in cui n = m, ottengo quello che è il polinomio interpolante. Visto che quello che ci interessa però è il polinomio approssimante p_n^*, considereremo solo il caso in cui n < m.

Per poter risolvere il problema di minimo esprimiamo il polinomio in funzione della base canonica, ottenendo

$$p_n(x_i) = \sum_{j=0}^{n} a_j x_i^j = (\mathrm{X}\mathbf{a})_i \ , \qquad \mathrm{i} = 0, \ \ldots \ \mathrm{m}$$

dove

- $\mathbf{a} = (a_0, a_1, \ \ldots \ , a_n)^T \in \boldsymbol{\mathcal{R}}^{n+1}$ è il vettore dei coefficienti del polinomio,

- $\mathrm{X} = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_m & x_m^2 & \cdots & x_m^n \end{bmatrix}$ è la matrice di Vandermonde di dimensioni (m+1) x (n+1).

Di conseguenza si ha che

$$\|p_n - \ f\|_2^2 = \sum_{i=0}^{m} [(\mathrm{Xa})_i - \ y_i]^2 = \|\mathrm{X}\mathbf{a} - \mathbf{y}\|_2^2. \qquad \text{(II.5)}$$

Nel caso specifico quindi, il problema di minimo assume la forma

$$\min_{p_n \in \Pi_n} \|\mathrm{X}\mathbf{a} - \mathbf{y}\|_2^2 \ , \qquad \text{(II.6)}$$

che non è altro che un problema ai minimi quadrati. Un metodo stabile ed efficiente per la sua risoluzione è utilizzare la fattorizzazione QR della matrice X. Data quindi la scomposizione

$$\mathrm{X} = \mathrm{QR}, \qquad \text{con} \qquad \mathrm{Q} \in \boldsymbol{\mathcal{R}}^{(m+1)\,x\,(m+1)} \ , \ \ \mathrm{R} \in \boldsymbol{\mathcal{R}}^{(m+1)\,x\,(n+1)},$$

e sostituendo nel problema di minimo, ottengo

$$\|\mathrm{X}\mathbf{a} - \mathbf{y}\|_2^2 = \|\mathrm{QR}\mathbf{a} - \mathbf{y}\|_2^2 = \|\mathrm{Q}(\mathrm{R}\mathbf{a} - \mathrm{Q}^T\mathbf{y})\|_2^2 = \|\mathrm{R}\mathbf{a} - \mathbf{c}\|_2^2 \qquad \text{(II.7)}$$

dove ho posto $\mathbf{c} = \mathrm{Q}^T\mathbf{y}$ e ho sfruttato il fatto che una matrice ortogonale non modifica la norma-2 di un vettore. La matrice R essendo una matrice rettangolare, può essere rappresentata come

$$\mathrm{R} = \begin{bmatrix} \mathrm{R}_1 \\ 0 \end{bmatrix},$$

con R_1 matrice triangolare superiore, quadrata e, se X è a rango pieno, non singolare. Partizionando anche il vettore $\mathbf{c}$ nello stesso modo si ha

$$\mathbf{c} = \begin{bmatrix} \mathbf{c}_1 \\ \mathbf{c}_2 \end{bmatrix}, \qquad \text{con} \quad \mathbf{c}_1 \in \boldsymbol{\mathcal{R}}^n \ , \quad \mathbf{c}_2 \in \boldsymbol{\mathcal{R}}^{m-m} \ ,$$

e, sostituendo le grandezze partizionate nel problema II.7 , arriviamo ad ottenere

$$\|\mathrm{R}\mathbf{a} - \mathbf{c}\|_2^2 = \left\| \begin{bmatrix} \mathrm{R}_1\mathbf{a} \\ 0 \end{bmatrix} - \begin{bmatrix} \mathbf{c}_1 \\ \mathbf{c}_2 \end{bmatrix} \right\|_2^2 = \|\mathrm{R}_1\mathbf{a} - \mathbf{c}_1\|_2^2 + \|\mathbf{c}_2\|_2^2 ,$$

e il problema di minimo diventa

$$\min_{\mathbf{a} \in \mathcal{R}^{n+1}} \|\mathrm{R}_1\mathbf{a} - \mathbf{c}_1\|_2^2 + \|\mathbf{c}_2\|_2^2 . \qquad \text{(II.8)}$$

Dalla (II.8) vediamo chiaramente che il vettore $\mathbf{a}$, vettore dei coefficienti di p_n^*, diventa la soluzione del problema ai minimi quadrati, nonché la soluzione del sistema lineare

$$\mathrm{R}_1\mathbf{a} = \mathbf{c}_1. \qquad \text{(II.9)}$$

Se il $\det(\mathrm{R}_1) \neq 0$, allora il sistema ammette una sola soluzione per la quale $\mathrm{R}_1\mathbf{a} - \mathbf{c}_1 = 0$, e che in corrispondenza della quale il minimo della (II.8) assume il valore

$$\min_{\mathbf{a} \in \mathcal{R}^{n+1}} \|\mathrm{X}\mathbf{a} - \mathbf{y}\|_2^2 = \|\mathbf{c}_2\|_2^2. \qquad \text{(II.10)}$$

Nel caso in cui $\|\mathbf{c}_2\|_2 = 0$, il vettore $\mathbf{a}$ è la soluzione classica del sistema iniziale $\mathrm{X}\mathbf{a} = \mathbf{y}$, mentre se $\|\mathbf{c}_2\|_2 \neq 0$, $\mathbf{a}$ risulta essere la soluzione nel senso dei minimi quadrati. In quest'ultimo caso, la norma di $\mathbf{c}_2$ risulta essere la misura del residuo. Nel calcolo del polinomio di miglior approssimazione ovviamente avremo che $\mathbf{c}_2 \neq 0$.

Abbiamo quindi ottenuto che, grazie alla fattorizzazione QR, il calcolo di p_n^* si riduce alla risoluzione di un sistema lineare triangolare superiore la cui soluzione risulta essere un vettore le cui componenti sono i coefficienti del polinomio approssimante. Il vantaggio nell'utilizzare la QR invece della fattorizzazione di Cholesky è dato dal minore condizionamento della matrice del sistema che, utilizzando quest'ultima fattorizzazione, risulterebbe il quadrato del condizionamento di (II.9). Ciò è dovuto al fatto che Cholesky necessita di una matrice simmetrica e definita positiva e questo mi avrebbe portato a dover risolvere il sistema normale

$$\mathrm{X}^T\mathrm{X}\,\mathbf{a} = \mathrm{X}^T\mathbf{y}$$

con un conseguente aumento degli errori nella soluzione.

II.2 Implementazione e Test

In questa sezione implementeremo il calcolo del polinomio approssimante sul software MatLab, così come lo abbiamo sviluppato matematicamente nella sezione precedente. Abbiamo scelto come funzioni test da approssimare, le funzioni

1) seno,
2) di Runge,
3) logaritmo naturale.

Tali funzioni verranno inizialmente campionate in base ad un numero specifico di ascisse prese equispaziate. I campioni così ottenuti verranno sporcati da un errore random di distribuzione gaussiana ottenuto mediante la funzione MatLab `randn()`e, il polinomio approssimante, verrà calcolato proprio in base ai campioni affetti da errore. Questo simula quelle situazioni in cui la funzione è conosciuta su base campionaria per esempio risultante da misure affette da errore, situazione in cui l'approssimazione risulta più performante dell'interpolazione. Dove necessario verranno inoltre calcolati l'errore di approssimazione (in norma-2) rispetto alla funzione vera o rispetto ai campioni affetti da errore. Di seguito riportiamo gli script utilizzati.

File II.1 `miglior approssimazione.m`

```
%Calcolo del polinomio di migliore approssimazione uniforme.
%Per la prova verranno scelte tre funzioni differenti per le quali verranno
%scelti i seguenti parametri:
%M = numero di ascisse su cui campionare le funzioni da approssimare
%n = grado del polinomio approssimante
%I dati verranno sporcati con degli errori a distribuzione gaussiana in
%modo da poter valutare la bontà dell'approssimazione. Per risolvere il
%problema ai minimi quadrati risultante verrà usata la fattorizzazione QR.

%Selezione della funzione da approssimare. la variabile select può assumere
%tre valori a seconda della funzione di cui si vuole testare
%l'approssimazione:
%select = 1   --> funzione seno
%select = 2   --> funzione di Runge
%select = 3   --> logaritmo naturale

disp('Approssimazione di funzioni ai minimi quadrati')
disp('seno    --> digitare 1')
disp('Runge   --> digitare 2')
disp('log(x)  --> digitare 3')
select = input('Selezionare la funzione da approssimare      <--  ');

if((select ~= 1) && (select ~= 2) && (select ~= 3))
    disp('Nessuna funzione con questo numero')
    return;
end

M = input('Inserire il numero di punti da campionare    M =  ');
e = input('Inserire il fattore di scala dell errore \nDeve essere un numero compreso tra
0.2 e 0.8 e =  ');

if((e < 0.2) || (e > 0.8))
    disp('Fattore di scala dell errore sbagliato')
    return;
end

n = input('Inserire il grado del polinomio approssimante \nDeve essere strettamente
```

```
minore di M-1, n =  ');

if(n >= M-1)
    disp('Grado del polinomio troppo elevato')
    return;
end

switch select
    case 1
       t = linspace(-1,1,M);
       b = sin(pi*t) + (e*randn(M,1))';
       x = linspace(-1,1,201);
       f = sin(pi*x);
    case 2
       h = input('Scegliere il parametro h per la funzione di Runge, h =  ');
       s = input('Scegliere il parametro h per la funzione di Runge, s =  ');
       t = linspace(-1,1,M);
       b = runge(t,h,s) + (e*randn(M,1))';
       x = linspace(-1,1,201);
       f = runge(x,h,s);
    case 3
       t = linspace(1,3,M);
       b = log(t) + (e*randn(M,1))';
       x = linspace(1,3,201);
       f = log(x);
    otherwise
       disp('La funzione selezionata non esiste')
       return;
end

%Calcolo del polinomio di migliore approssimazione.
%Costruzione della matrice di Vandermonde.
X = zeros(M,n+1);
for i = 1 : M
    for j = 1 : n+1
        X(i,j) = (t(i))^(j-1);
    end
end

%Risoluzione del problema ai minimi quadrati risultante, min||Xa - b||.
Q = zeros(M,M);
R = zeros(M,n+1);
[Q,R] = myQR(X);
c = Q'*b';
a = utss(R(1:n+1,1:n+1),c(1:n+1));

%Costruzione del polinomio approssimante su tutte le 201 ascisse.
P = zeros(201,n+1);
for k = 1 : 201
    for l = 1 : n+1
        P(k,l) = (x(k))^(l-1);
    end

    p(k) = P(k,:)*a';
end

%Polinomio approssimante sulle sole ascisse di campionamento.
pn = X*a';

error = pn - b';
leg1 = 1;
leg2 = 1;

%Plot delle funzioni.
subplot(2,1,1),
if(select ~= 3)
    plot(t,b,'ro',x,f,'k-',x,p,'g-');
else
    semilogy(t,b,'ro',x,f,'k-',x,p,'g-');
```

```
end
if(select == 1)
    leg1 = 2;
    axis([-1,1,-3,3]);
end
if(select == 2)
    leg1 = 1;
    axis([-1,1,-h*2,h*2]);
end
if(select == 3)
    leg1 = 4;
    axis([1,3,-3,3]);
end
legend('Funzione Sporca','Funzione','Polinomio',leg1);
title('Polinomio di migliore Approssimazione');
xlabel('Dominio di Campionamento');
ylabel('Ampiezze');

%Errore sulle ascisse di campionamento.
subplot(2,1,2),
plot(t,error,'go');
grid on
if(select == 1)
    leg2 = 2;
    axis([-1,1,-3,3]);
end
if(select == 2)
    leg2 = 2;
    axis([-1,1,-h*2,h*2]);
end
if(select == 3)
    leg2 = 2;
    axis([1,3,-3,3]);
end
legend('Errore Totale',leg2);
title('Errore di approssimazione rispetto ai campioni della funzione');
xlabel('Dominio di Campionamento');
ylabel('Ampiezze');
```

Una volta che il file II.1 viene lanciato, questo permette di scegliere la funzione da approssimare, il numero di punti su cui campionare M, il fattore di scala dell'errore, il grado del polinomio approssimante n e, per la funzione di Runge, i parametri s e h che ne determinano la forma. Tale script calcola il polinomio approssimante e plotta la funzione scelta insieme al polinomio trovato. Il file II.2 permette le stesse scelte del precedente tranne per il fatto che stavolta viene inserito il grado massimo del polinomio approssimante. Infatti lo script calcola i polinomi approssimanti di ogni grado, a partire dal grado 0 sino al grado massimo m inserito. Questo viene fatto per poter stabilire qual'è l'errore in funzione del grado del polinomio. Vengono calcolati due errori, quello rispetto alla funzione f, che risulta essere l'errore di estrapolazione e quello rispetto alle ascisse di campionamento che risulta essere il vero e proprio errore di approssimazione.

File II.2 errore di approssimazione.m

```
%Calcolo dell' errore di approssimazione in funzione del grado del polinomio
%approssimante.

%Selezione della funzione da approssimare. la variabile select può assumere
%tre valori a seconda della funzione di cui si vuole testare
%l'approssimazione:

%select = 1   --> funzione seno
%select = 2   --> funzione di Runge
%select = 3   --> logaritmo naturale
```

```
disp('Approssimazione di funzioni ai minimi quadrati')
disp('seno    --> digitare 1')
disp('Runge   --> digitare 2')
disp('log(x)  --> digitare 3')
select = input('Selezionare la funzione da approssimare      <--  ');

if((select ~= 1) && (select ~= 2) && (select ~= 3))
    disp('Nessuna funzione con questo numero')
    return;
end

M = input('Inserire il numero di punti da campionare    M =  ');
e = input('Inserire il fattore di scala dell errore \nDeve essere un numero compreso tra
0.2 e 0.8 e =  ');

if((e < 0.2) || (e > 0.8))
    disp('Fattore di scala dell errore sbagliato')
    return;
end

n = input('Inserire il grado max del polinomio approssimante \nDeve essere strettamente
minore di M-1,        n =  ');

if(n >= M-1)
    disp('Grado max del polinomio troppo elevato')
    return;
end

switch select
    case 1
       t = linspace(-1,1,M);
       b = sin(pi*t) + (e*randn(M,1))';
       x = linspace(-1,1,201);
       f = sin(pi*x);
    case 2
       h = input('Scegliere il parametro h per la funzione di Runge, h =  ');
       s = input('Scegliere il parametro h per la funzione di Runge, s =  ');
       t = linspace(-1,1,M);
       b = runge(t,h,s) + (e*randn(M,1))';
       x = linspace(-1,1,201);
       f = runge(x,h,s);
    case 3
       t = linspace(1,3,M);
       b = log(t) + (e*randn(M,1))';
       x = linspace(1,3,201);
       f = log(x);
    otherwise
       disp('La funzione selezionata non esiste')
       return;
end

error1 = 0;
error2 = 0;

for m = 0 : n
    %Calcolo del polinomio di migliore approssimazione.
    %Costruzione della matrice di Vandermonde.
    X = zeros(M,m+1);
    for i = 1 : M
        for j = 1 : m+1
            X(i,j) = (t(i))^(j-1);
        end
    end

    %Risoluzione del problema ai minimi quadrati risultante, min||Xa - b||.
    Q = zeros(M,M);
    R = zeros(M,m+1);
    [Q,R] = myQR(X);
```

```
    c = Q'*b';
    a = utss(R(1:m+1,1:m+1),c(1:m+1));

    %Costruzione del polinomio approssimante su tutte le 201 ascisse.
    P = zeros(201,m+1);
    for k = 1 : 201
        for l = 1 : m+1
            P(k,l) = (x(k))^(l-1);
        end

        p(k) = P(k,:)*a';
    end

    %Polinomio approssimante sulle sole ascisse di campionamento.
    clearvars pn
    pn = X*a';


    err1 = pn - b';
    err2 = p  - f;

    error1(m+1) = norm(err1);
    error2(m+1) = norm(err2);

    degre(m+1)  = m;

end

%Plot dell'errore in funzione del grado del polinomio.
semilogy(degre,error1,'g-',degre,error2,'k-');
legend('Errore sui campioni','Errore sulla funzione',2);
title('Errore di approssimazione in funzione del grado del polinomio');
xlabel('n [Grado del polinomio approssimante]');
ylabel('error [Errore di approssimazione]');
```

II.2.1 Funzione Seno

Il test sulla funzione sen(πx) è stato condotto con i seguenti valori dei parametri
1) M = 50,
2) e = 0.3,

e per il grado del polinomio abbiamo scelto tre valori possibili
3) n = 0,
4) n = 10,
5) n = 20.

In figura II.1 vediamo riportati i risultati della prima prova. Si può notare che il polinomio approssimante di grado 0 (linea verde) è una retta parallela all'asse delle ascisse, di equazione y = -0.0216 che in sostanza non è altro che il valore medio della funzione seno stessa. Infatti il polinomio approssimante non fa altro che mediare i campioni con cui viene calcolato. Nel caso sia di grado 0, è una retta, e in questo caso risulta essere praticamente il valore medio del seno. Questo ci viene confermato anche dal plot degli errori sui campioni sporchi che risultano avere una distribuzione abbastanza simmetrica rispetto all'asse delle ascisse.

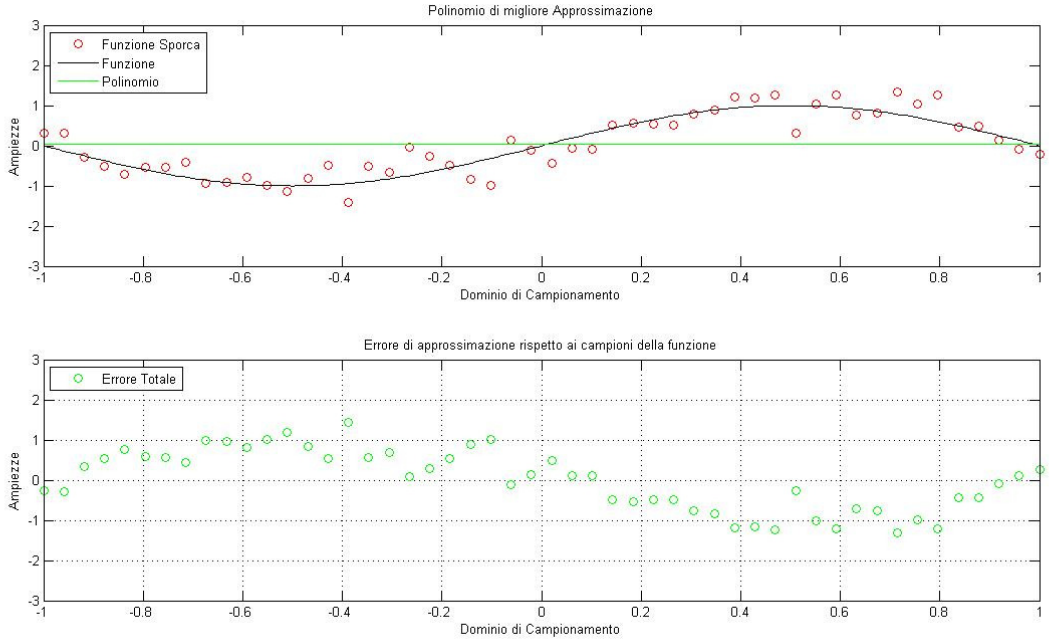

Figura II.1: Approssimazione della funzione seno con un polinomio di grado n = 0 e relativo errore.

Aumentando il grado ad n = 10, abbiamo un polinomio che segue abbastanza bene la funzione seno stessa e che riesce a mediare molto meglio i campioni con cui è stato calcolato (cerchi rossi figura II.2). Osservando anche gli errori commessi sulle ascisse di campionamento, vediamo infatti che sono diminuiti in quanto si sono raggruppati maggiormente intorno all'asse delle ascisse. Portando il grado del polinomio ad n = 20 invece, abbiamo la comparsa di oscillazioni. Aumentando il grado infatti il polinomio cerca di seguire maggiormente i campioni con conseguente perdita graduale delle caratteristiche di mediazione (se portassimo il grado ad n = 49, avremmo un polinomio interpolante che attraverserebbe tutti i campioni, con un errore di estrapolazione però molto alto).

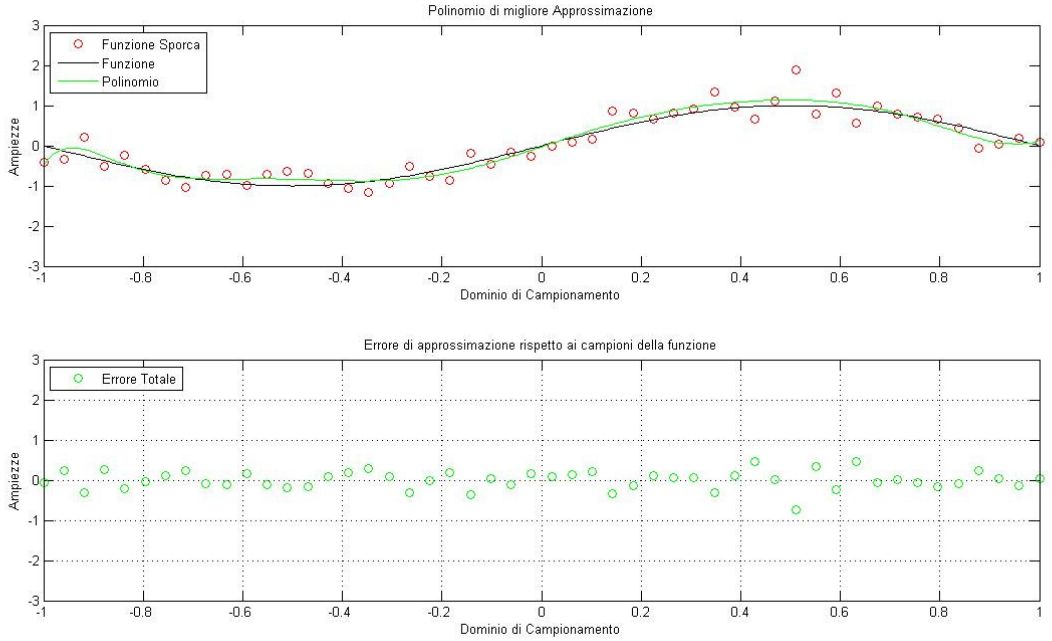

Figura II.2: Approssimazione della funzione seno con un polinomio di grado n = 10 e relativo errore.

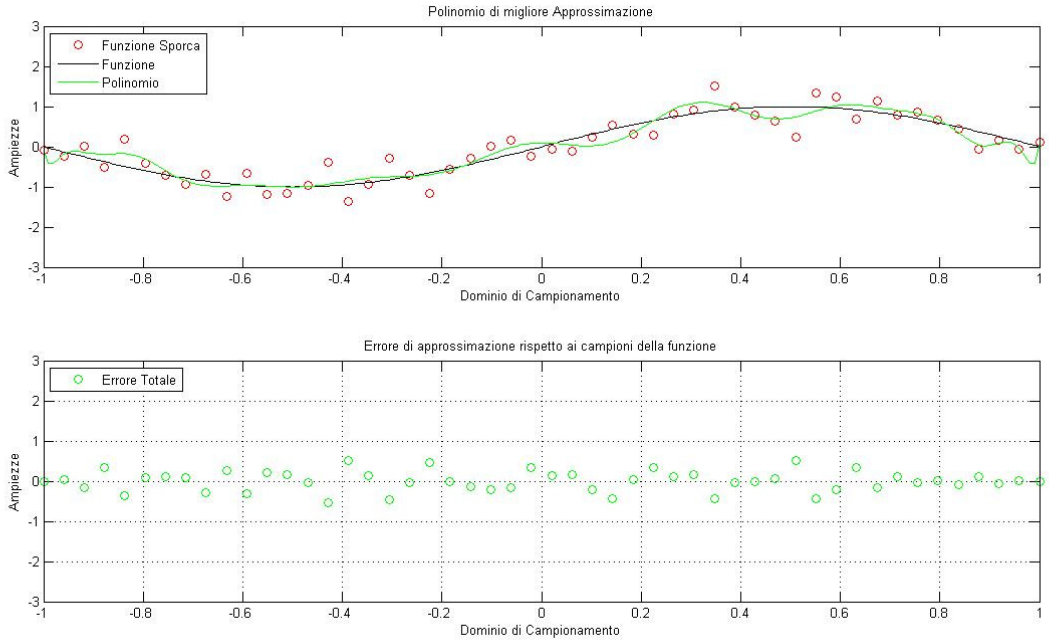

Figura II.3: Approssimazione della funzione seno con un polinomio di grado n = 20 e relativo errore.

In figura II.4 abbiamo usato un polinomio di grado n = 30, con un conseguente aumento delle oscillazioni ma con una diminuzione ulteriore dell'errore di approssimazione.
Quanto detto può essere osservato anche in figura II.5 dove sono plottati gli errori di approssimazione e di estrapolazione del seno, in funzione del grado del polinomio. La prova è stata condotta con gli stessi valori dei parametri usati precedentemente e con grado variante da n = 0 a n = 40. Si può notare che mentre l'errore di approssimazione sui campioni sporchi diminuisce all'aumentare di n, l'errore di estrapolazione al di fuori delle ascisse di

campionamento al contrario aumenta a causa delle oscillazioni crescenti. Una possibile scelta per il grado da assegnare al polinomio di approssimazione và da n = 10 a n =12, dove i due errori pressoché coincidono prima di divergere definitivamente.

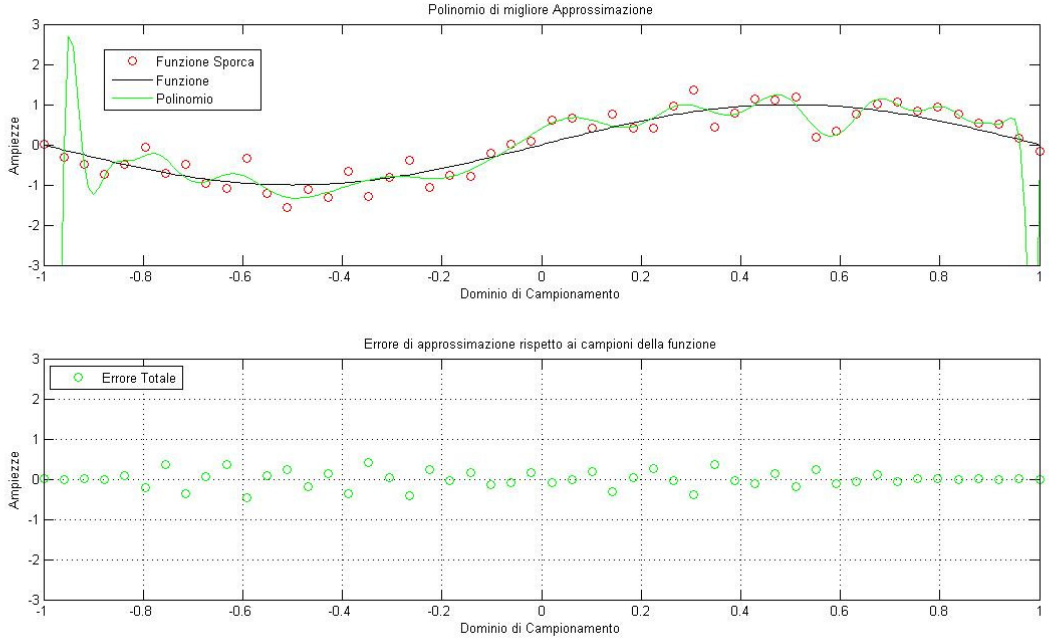

Figura II.4: Approssimazione della funzione seno con un polinomio di grado n = 30 e relativo errore.

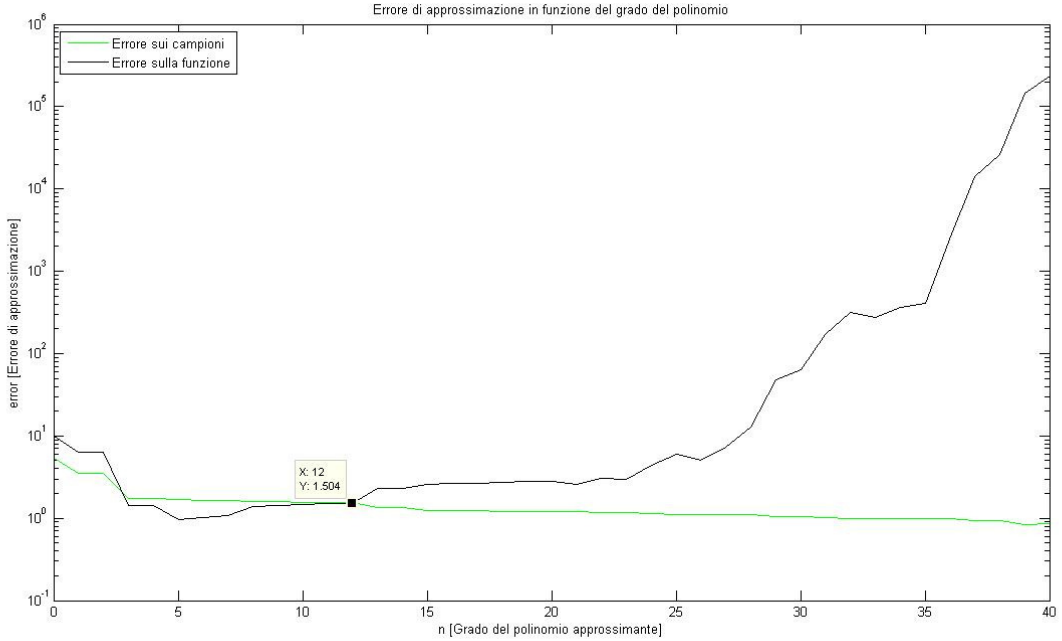

Figura II.5: Errore nell'approssimazione della funzione seno con polinomi di grado crescente.

II.2.2 Funzione di Runge

Per effettuare questo test è stata scritta una function, la `runge.m`, che implementa appunto la funzione di Runge.

La prima prova è stata condotta con i seguenti valori dei parametri

1) M = 30,
2) e = 0.3,
3) n = 0,
4) h = 1,
5) s = 25.

I risultati mostrano ancora una volta la capacità del polinomio di approssimazione di mediare i campioni in base ai quali viene calcolato. Infatti, il polinomio di grado 0 non è altro che una retta parallela all'asse delle ascisse, di equazione y = 0.3260 (figura II.6). Se andiamo ad effettuare la media dei valori del vettore b, ovvero del vettore dei campioni sporcati usato per il calcolo, otteniamo esattamente il coefficiente del polinomio di grado 0,

```
d = mean(b)

d =

    0.3260
```

confermando così quanto visto anche per la funzione seno.

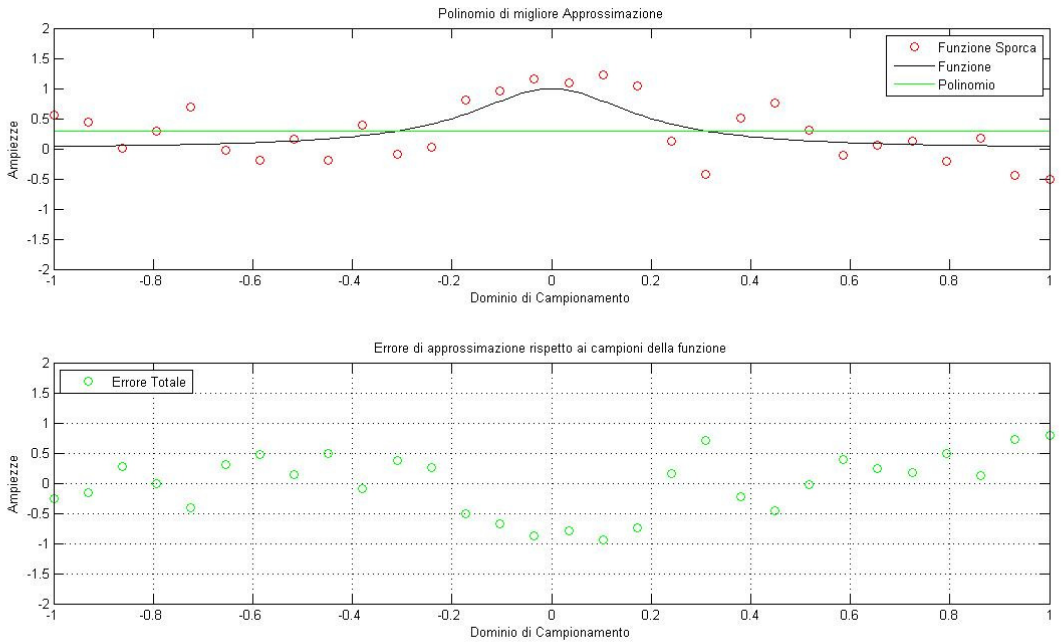

Figura II.6: Approssimazione della funzione di Runge con un polinomio di grado n = 0 e relativo errore.

Quello che vogliamo verificare è l'effetto provocato dall'aumento del numero delle ascisse di campionamento, tenendo fisse tutte le altre grandezze. Effettuiamo quindi una prova con i seguenti parametri

1) $M = 20$,
2) $e = 0.3$,
3) $n = 14$,
4) $h = 1$,
5) $s = 25$.

In figura II.7 sono riportato i risultati. Vediamo che il polinomio approssimante presenta già delle leggere oscillazioni che si accentuano ai lati, essendo comunque il suo grado vicino al numero di punti campionari. Dal grafico notiamo infatti che molti dei punti vengono attraversati da esso e questo si rispecchia anche nel grafico dell'errore che mostra come l'errore di approssimazione rispetto ai campioni sia molto basso (i punti sono raccolti vicino all'asse delle ascisse).

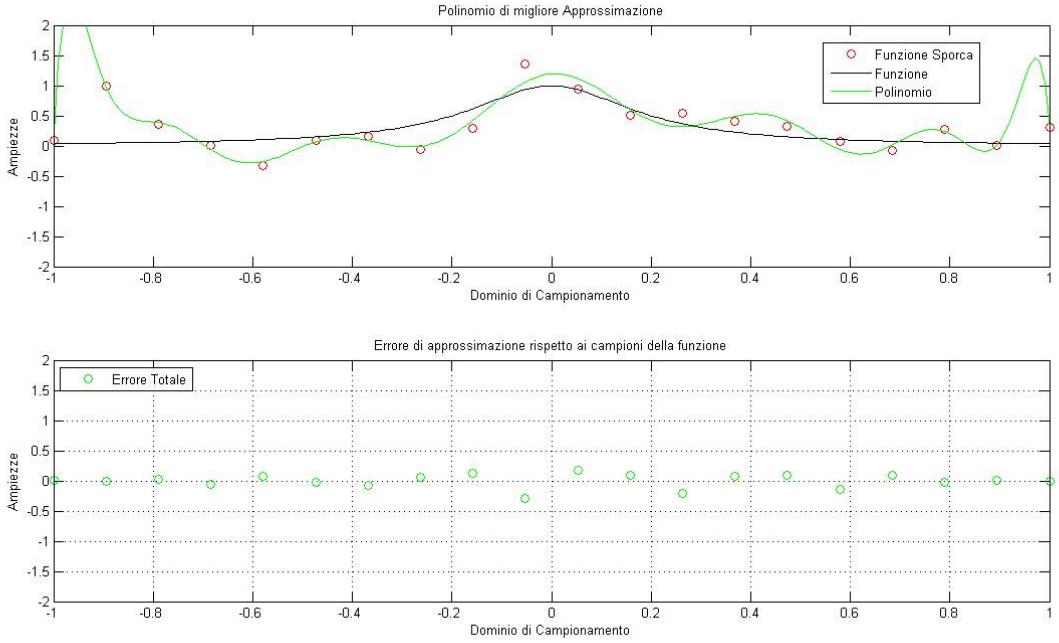

Figura II.7: Approssimazione della funzione di Runge con un polinomio di grado $n = 14$ e $M = 20$ punti campionari.

Aumentando il numero dei campioni, vediamo che le oscillazioni tendono ad addolcirsi. In figura II.8 abbiamo ripetuto la stessa prova portando il numero di campioni da 20 a 30. Notiamo immediatamente che le forti oscillazioni presenti agli estremi del dominio hanno lasciato il posto a delle variazioni molto più smorzate. Aumentando M da 30 a 50 abbiamo che questo effetto è ancora più accentuato. In figura II.9 possiamo verificare infatti che il polinomio approssimante oscilla ancora meno di prima, ma parallelamente è aumentato l'errore di approssimazione rispetto ai campioni della funzione. In pratica stiamo osservando quanto abbiamo visto con la funzione seno quando abbiamo gradualmente aumentato il grado del polinomio. Man mano che aumenta la differenza tra il numero dei campioni e il grado del polinomio, quest'ultimo si comporta sempre meno da interpolatore e sempre più da approssimatore, con conseguente aumento dell'errore di approssimazione sui campioni. Le figure II.10 e II.11, realizzate con 90 e 120 campioni, confermano quanto detto.

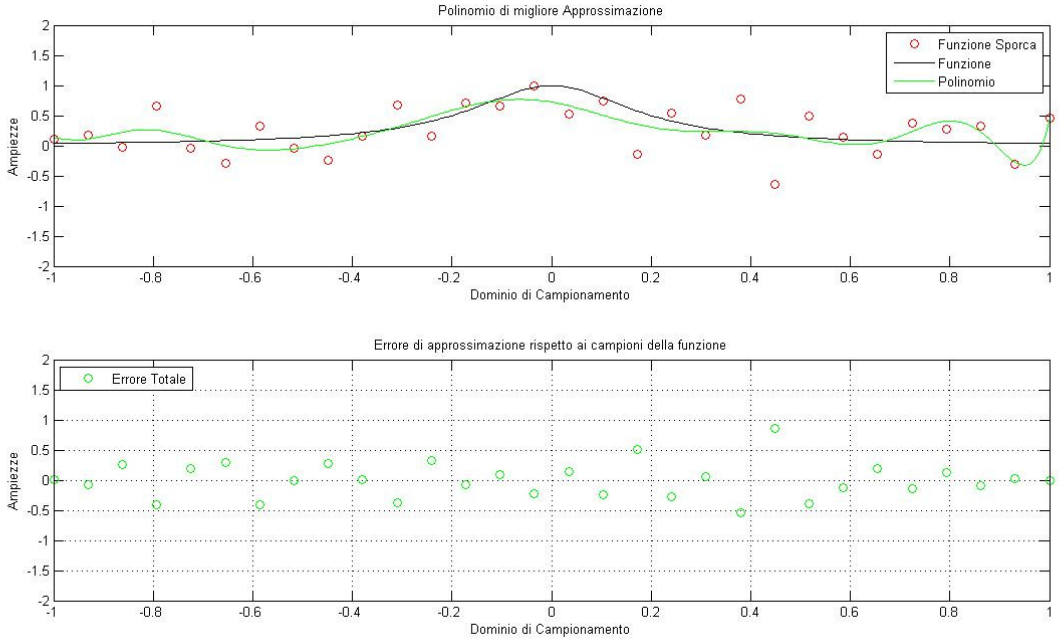

Figura II.8: Approssimazione della funzione di Runge con un polinomio di grado n = 14 e M = 30 punti campionari.

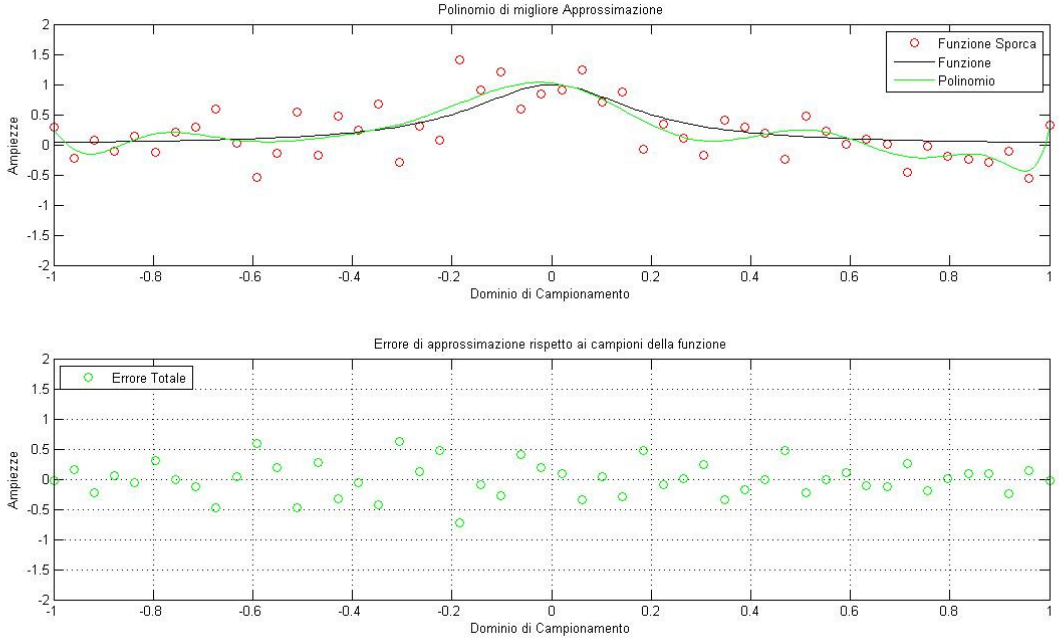

Figura II.9: Approssimazione della funzione di Runge con un polinomio di grado n = 14 e M = 50 punti campionari.

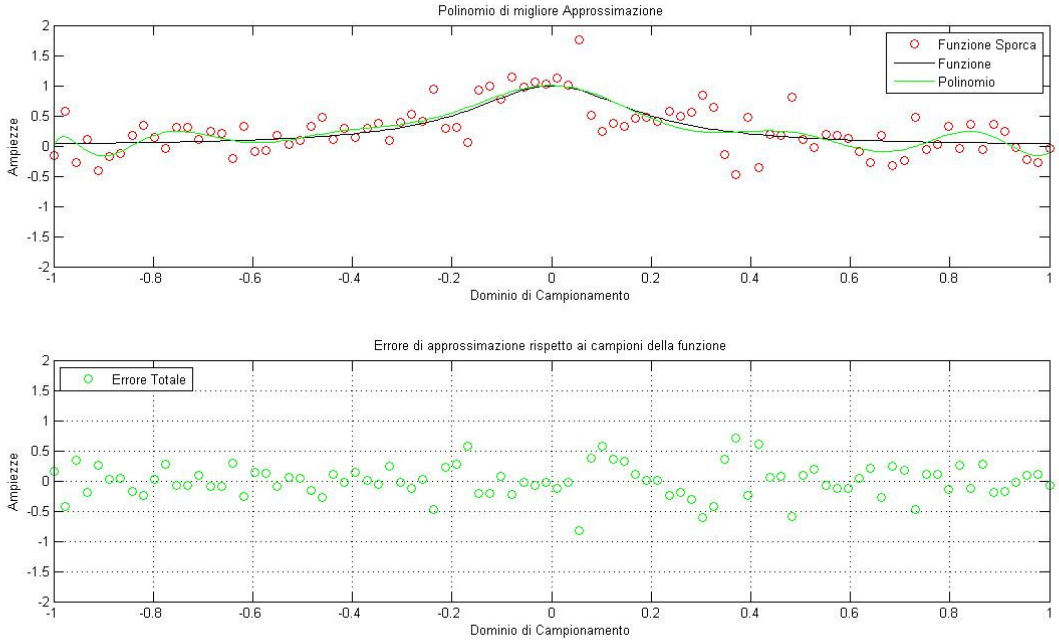

Figura II.10: Approssimazione della funzione di Runge con un polinomio di grado n = 14 e M = 90 punti campionari.

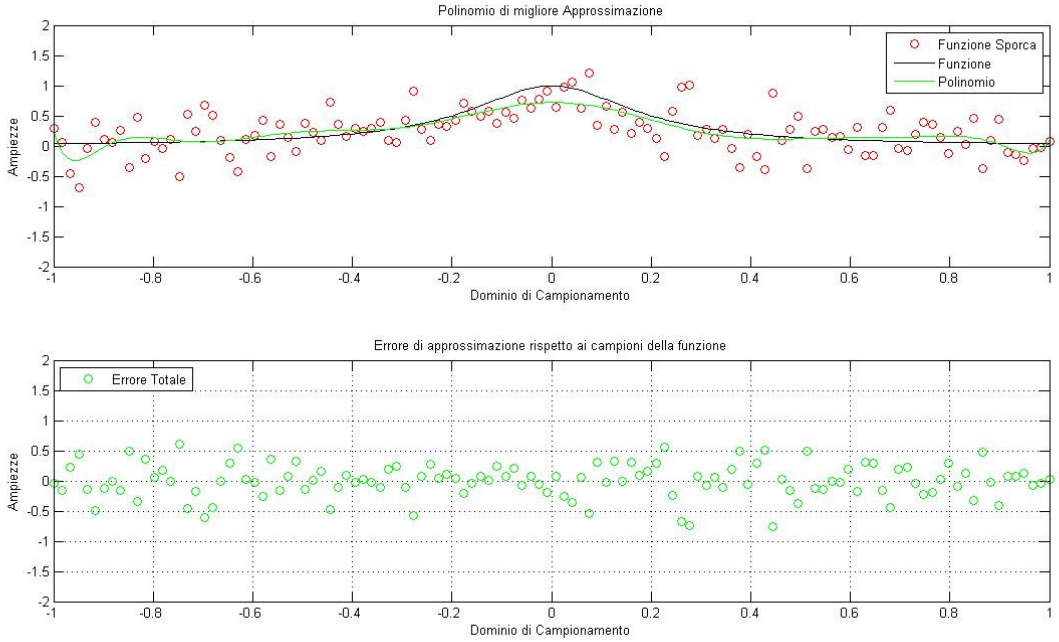

Figura II.11: Approssimazione della funzione di Runge con un polinomio di grado n = 14 e M = 120 punti campionari.

II.2.3 Funzione Logaritmo

Per quanto riguarda la funzione logaritmo, abbiamo scelto di effettuare delle prove variando solo l'ampiezza dell'errore presente nei campioni utilizzati per la costruzione del polinomio approssimante. Partiamo dai seguenti valori dei parametri e plottiamo i risultati

1) $M = 25$,
2) $e = 0.2$,
3) $n = 5$.

In figura II.12 osserviamo che il polinomio riesce ad approssimare in maniera abbastanza buona la funzione logaritmo e che l'errore sui campioni affetti da errore è abbastanza basso. In figura II.13 invece vediamo il risultato della seconda prova, con valore del fattore di scala dell'errore di 0.4. Vediamo la comparsa di oscillazioni e di una deviazione nella prima parte della curva. Nel grafico dell'errore vediamo già che questo aumenta, infatti i pallini verdi si sono allontanati dall'asse delle ascisse. In figura II.13 vediamo il risultato della prova condotta con $e = 0.6$ e in figura II.14 il risultato della prova con $e = 0.8$. Come previsto, all'aumentare dell'errore sono aumentate le oscillazioni nel polinomio approssimante che, cercando di mediare il valore dei campioni con cui viene calcolato, si discosta sempre di più dalla funzione da approssimare. Parallelamente vediamo come aumenti anche l'errore di approssimazione, con i pallini verdi che si allontanano sempre più dall'asse delle ascisse.

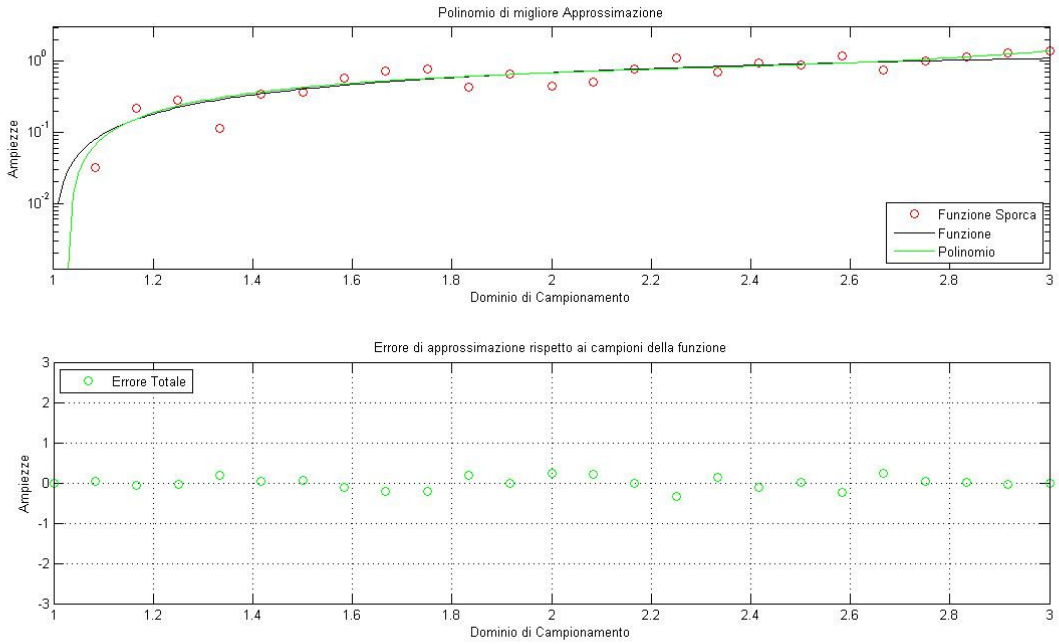

Figura II.12: Approssimazione del logaritmo naturale con un polinomio di grado $n = 5$ e un errore $e = 0.2$.

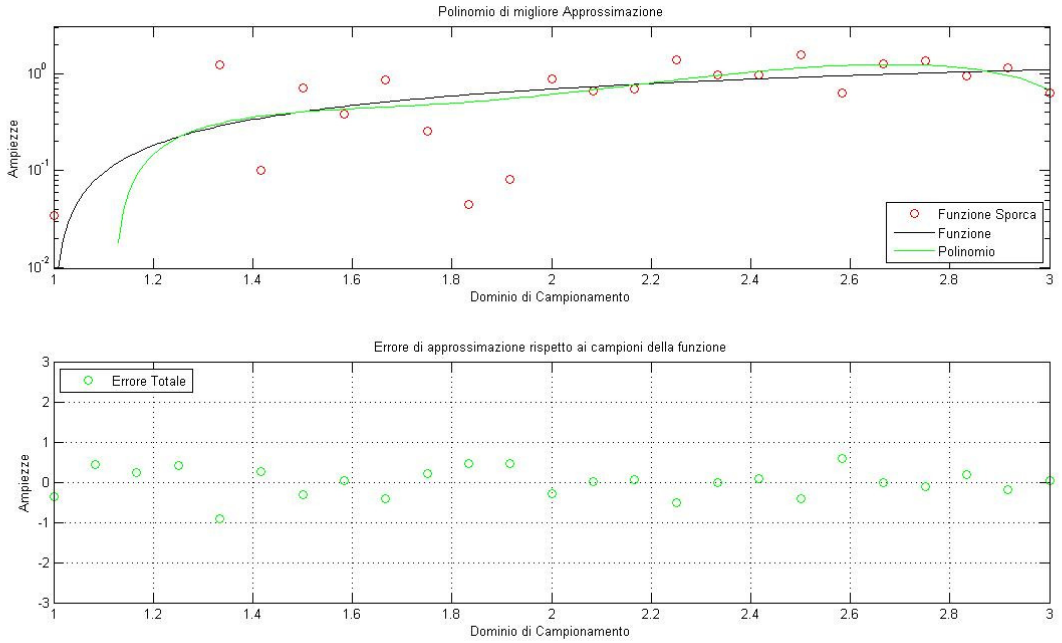

Figura II.13: Approssimazione del logaritmo naturale con un polinomio di grado n = 5 e un errore e = 0.4.

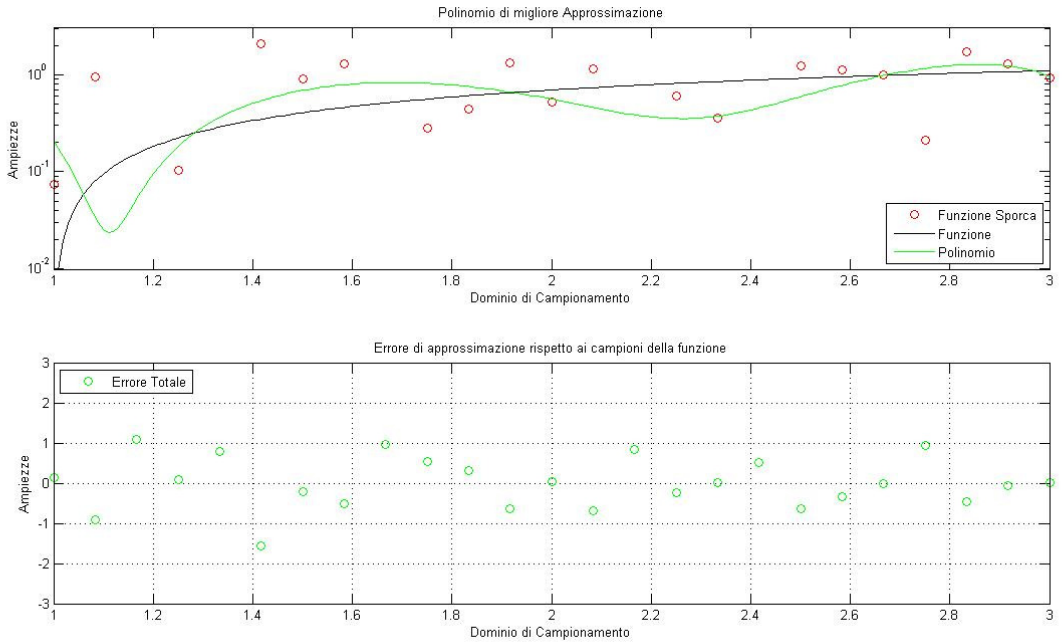

Figura II.14: Approssimazione del logaritmo naturale con un polinomio di grado n = 5 e un errore e = 0.6.

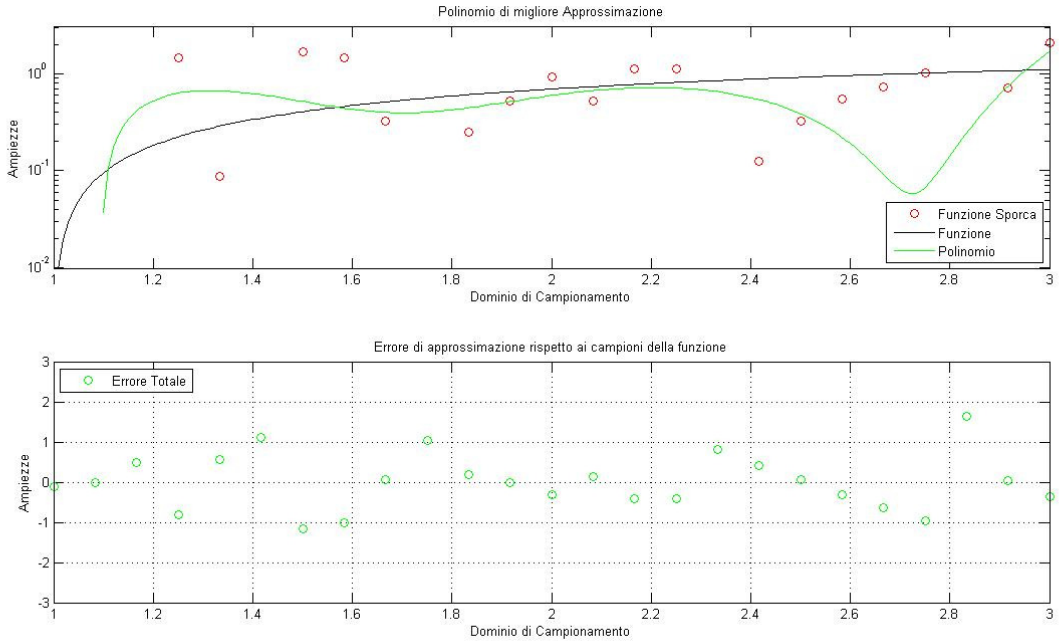

Figura II.15: Approssimazione del logaritmo naturale con un polinomio di grado n = 5 e un errore e = 0.8.

Appendice A - Matrici test

I test sulle fattorizzazioni QR e di Cholesky, effettuati nella prima parte, vedono coinvolte due tipi differenti di matrici. Il primo tipo è del tutto generale ed è una semplice matrice di numeri **random** mentre la seconda è la ben nota matrice di **Hilbert**. Entrambe le matrici hanno caratteristiche differenti e sono adatte a differenti prove.

Una matrice **random** è una matrice, in generale rettangolare, costituita da numeri random ottenuti mediante una predeterminata distribuzione di probabilità. In generale tale matrice non è ne simmetrica ne definita positiva, per cui per i test sulla fattorizzazione di Cholesky e nei test in cui quest'ultima viene confrontata con la QR, viene utilizzata per costruire una matrice B tale che data una matrice random A, la matrice $B = A^T A$ sia ancora random e abbia anche le caratteristiche sopraccitate. In MatLab ci sono diverse istruzioni per ottenere matrici di questo tipo, tramite diverse distribuzioni. Quella che noi abbiamo usato è l'istruzione `rand()` che genera una matrice di numeri casuali uniformemente distribuiti nell'intervallo aperto (0, 1). Un sample del codice utilizzato può essere il seguente:

```
m = 5;
n = 4;
A = rand(m,n);
B = A'*A;
```

Il problema fondamentale di utilizzare una matrice così costruita è che il condizionamento aumenta in maniera esponenziale. Infatti per le proprietà del numero di condizionamento si ha che:

$$\boldsymbol{k}(A) = \boldsymbol{k}(A^T)$$
$$\boldsymbol{k}(A^T A) = \boldsymbol{k}(A)\ \boldsymbol{k}(A^T) = \boldsymbol{k}(A)^2$$

Da cui vediamo chiaramente che la matrice B ha un condizionamento che risulta il quadrato di quello di A.

Una matrice di **Hilbert** è una matrice quadrata, simmetrica e definita positiva che è conosciuta tra le altre cose per essere mal condizionata. Gli elementi di tale matrice sono definiti dalla seguente relazione:

$$G_{ij} = \frac{1}{i+j+1} \qquad i,j = 1, \ \ldots \ , n$$

da cui vediamo che tali elementi sono tutti compresi tra 0 e 1. Inoltre, da tale espressione si vede anche che all'aumentare delle dimensioni della matrice gli elementi diminuiscono di valore, contribuendo così all'instabilità numerica della stessa. In MatLab una matrice di Hilbert di dimensione n può essere generata con l'istruzione `hilb()`:

```
n = 5;
H = hilb(n);
```

Possiamo confrontare l'andamento del condizionamento delle diverse matrici in funzione delle loro dimensioni mediante un semplice script. A tal fine è stato creato il file `condizionamento_matrici.m` dove abbiamo usato il condizionamento in norma-2 e di cui riportiamo il risultato grafico.

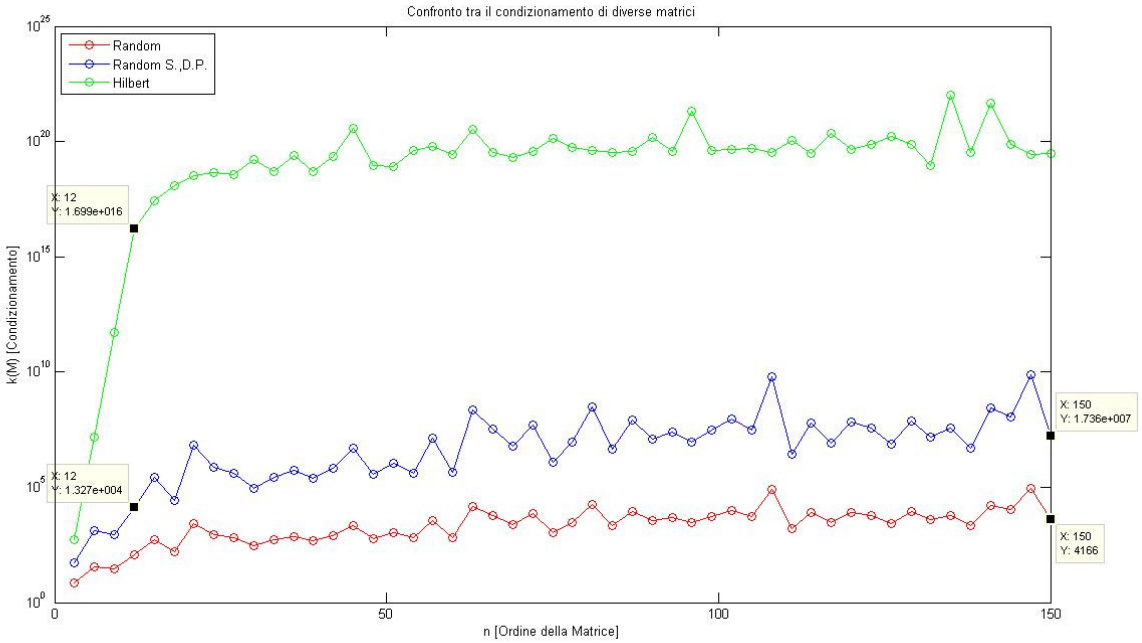

Figura A.1: Confronto tra il condizionamento di diverse matrici.

La figura mostra chiaramente che la matrice di Hilbert ha una propagazione degli errori intrinseca di gran lunga superiore rispetto alle altre due. Infatti, già ad una dimensione pari a 12, il numero di condizionamento è dell'ordine di 10^{16}, mentre le altre due matrici hanno un condizionamento di almeno 12 ordini di grandezza più piccolo. Il risultato di questa differenza lo si può vedere nell'utilizzo delle due funzioni `pdm1.m` e `pdm2.m` (positive definite matrix). Tali funzioni non fanno altro che determinare se una matrice è definita positiva o meno, sia tramite il calcolo dei minori principali (per matrici simmetriche – `pdm1.m`), sia tramite lo studio del segno degli autovalori (`pdm1.m`, `pdm2.m`). In entrambi i casi le matrici random non hanno dato nessun problema al variare dell'ordine (test compiuto sino ad $n = 200$) mentre per la matrice di Hilbert non si è riusciti a superare un ordine pari a 13 nel caso di `pdm1.m` (calcolo dei minori principali) e un ordine pari a 12 nel caso di `pdm2.m` (studio degli autovalori) senza avere non-riconoscimenti. Per tale ragione, avendo usato matrici test definite positive per l'algoritmo di Cholesky, nella sua implementazione è stato omesso un controllo per tale proprietà. Verifichiamo inoltre, che in media, il condizionamento delle due matrici random è uno il quadrato dell'altro così come previsto, come si vede per $n = 150$.

Appendice B - Matrici elementari di Householder

Una matrice elementare di Householder reale è una matrice che viene definita come:

$$H = I - 2ww^T \tag{B.1}$$

Dove:

- $w \in \mathcal{R}^n$ è un vettore la cui norma-2 è unitaria, $\|w\| = 1$.

La matrice così definita risulta essere, oltre che quadrata, anche simmetrica e ortogonale. Per costruire tale matrice andiamo a ricavare il vettore w in modo tale che sia verificata la relazione:

$$Hx = ke_1, \tag{B.2}$$

dove $k \in \mathcal{R}$ è una costante e e_1 è il primo versore della base canonica di $\mathcal{R}^n$ (nonché la prima colonna della matrice identità se vogliamo). Sostituiamo nella (B.2) l'espressione di H, ottenendo

$$Hx = x - 2ww^Tx = x - 2(w^Tx)w = ke_1,$$

da cui arriviamo all'espressione di w

$$w = \frac{x - ke_1}{2w^Tx}. \tag{B.3}$$

Ricordando però che il vettore w ha modulo unitario, allora la (B.3) può anche essere scritta nella forma:

$$w = \frac{x - ke_1}{\|x - ke_1\|}. \tag{B.4}$$

La costante k rimane determinata in modulo notando che, essendo la matrice H ortogonale, partendo dalla (B.2) e applicando la norma-2 ad ambo i membri abbiamo

$$\|Hx\| = \|x\| = |k| .$$

Quindi, ponendo $\|x\| = \sigma$, otteniamo

$$k = \pm\sigma , \tag{B.5}$$

espressione che mi lascia k libera in segno. Sino a qui, abbiamo così ottenuto i parametri necessari per il calcolo di H.

Per quanto riguarda il calcolo effettivo della matrice di Householder, alcuni accorgimenti portano a delle modifiche che risultano essere numericamente più convenienti, alleggerendo il peso computazionale del calcolo di w. Partendo infatti dalla relazione

$$x^T H\, x = x^T x - 2(w^T x)^2 = kx_1$$

otteniamo

$$(w^T x)^2 = \frac{\sigma^2 - kx_1}{2}\,.$$

Ricordando l'espressione (B.4), espressione di w trovata precedentemente, e mettendola in relazione a quest'ultima, otteniamo così

$$\|x - ke_1\| = 2w^T x = \sqrt{2(\sigma^2 - kx_1)}\,. \qquad (B.6)$$

Dalla (B.6) notiamo che quello che effettivamente viene migliorata numericamente è la normalizzazione di w.

Dal punto di vista numerico però l'espressione (B.6) può causare problemi in termini di possibile cancellazione, essendo questa presente al denominatore di un'altra espressione. Per evitare ciò, possiamo scegliere il segno di k, sino ad ora arbitrario, in modo tale che $-kx_1$ risulti sempre positivo. Per far ciò basterà porre

$$k = -\text{sign}(x_1)\sigma\,,$$

ottenendo

$$\|x - ke_1\| = \sqrt{2\sigma(\sigma + |x_1|)}\ . \qquad (B.7)$$

L'algoritmo così trovato viene riassunto nella seguente tabella.

ALGORITMO B.1 **Costruzione di una matrice elementare di Householder.**
1. $\sigma = \|x\|$ 2. $k = -\text{sign}(x_1)\sigma$ 3. $\lambda = \sqrt{2\sigma(\sigma + \|x_1\|)}$ 4. $w = x - ke_1/\lambda$ 5. $H = I - 2ww^T$

Possiamo introdurre un' ulteriore modifica alla costruzione di H cercando di eliminare la radice quadrata e la normalizzazione nel calcolo di w. Questo passaggio risulta utile per la riduzione degli errori introdotti dall'algoritmo stesso. Riprendendo le espressioni calcolate, possiamo notare che

$$2ww^T = \frac{(x-ke_1)(x-ke_1)^T}{\sigma(\sigma+|x_1|)}$$

e ponendo

$$v = x - ke_1$$

$$\beta = \sigma(\sigma + |x_1|)$$

otteniamo la seguente espressione per H

$$H = I - \frac{1}{\beta} vv^T \quad . \tag{B.8}$$

L'algoritmo modificato viene riassunto nella seguente tabella.

ALGORITMO B.2 **Costruzione di una matrice elementare di Householder (algoritmo modificato).**
1. $\sigma = \|x\|$ 2. $k = -\text{sign}(x_1)\sigma$ 3. $v = x - ke_1$ 4. $\beta = \sigma(\sigma + |x_1|)$ 5. $H = I - vv^T/\beta$

Appendice C - Indice dei Files MatLab

Elenchiamo di seguito i files MatLab nell'ordine con cui vi si è fatto riferimento nel testo.