Dispense ed esercizi per il Laboratorio di Calcolo Numerico

Elena Gaburro, elenagaburro@gmail.com

AA 2016-2017

Questi appunti non hanno alcuna pretesa di completezza. Sono solo alcune note ed esercizi che affiancano il corso di Laboratorio di Calcolo Numerico. Sono inoltre da considerarsi in perenne "under revision" e pertanto possono contenere discrepanze, inesattezze o errori.

Gli esercizi proposti durante l'esame scritto saranno ispirati a quelli presentati in queste pagine: alcune parti potranno essere simili, altre saranno delle variazioni. Potrà essere richiesto di saper commentare l'algoritmo implementato e i risultati ottenuti. E' permesso portare con sé dispense, appunti e i propri codici durante la prova scritta in laboratorio (è quindi vivamente consigliato lo svolgimento di tutti gli esercizi con commenti dettagliati!).

Le parti di dispensa scritte in questo colore rappresentano le modifiche rispetto alle dispense degli scorsi anni. Non escludo però che qualche (piccola) modifica sia scritta senza cambiare colore al testo.

Indice

Capitolo 1

Introduzione a MATLAB

Il nome MATLAB è acronimo di MATrix LABoratory. Originariamente MATLAB è stato sviluppato come ambiente interattivo per il calcolo matriciale ad alto livello. La principale caratteristica è che non opera con numeri, ma con *matrici*: i vettori e i numeri sono considerati casi particolari di matrici.

Attualmente MATLAB è utilizzato anche come calcolatrice scientifica evoluta, ambiente grafico, e linguaggio di programmazione.

1.1 Linguaggi interpretati e linguaggi compilati

I primi programmi per calcolatori elettronici sono stati scritti direttamente in codice macchina, nella forma di sequenze di bit riconoscibili dal calcolatore come istruzioni, ma molto rapidamente i programmatori hanno riconosciuto l'esigenza di definire linguaggi di programmazione più facili da capire e da utilizzare da parte degli umani. Le istruzioni numeriche sono state quindi sostituite da istruzioni scritte in modo più vicino al linguaggio naturale. Chiaramente, tali linguaggi non sono immediatamente comprensibili al calcolatore che quindi non è in grado di eseguirne direttamente le istruzioni. Per superare questo problema, sono state ideate due soluzioni.

La prima soluzione è quella di trasformare i programmi in codice comprensibile dal calcolatore *prima* di tentarne l'esecuzione. Questo processo di trasformazione si chiama *compilazione* e viene effettuato con l'ausilio di un programma che si chiama *compilatore*. La seconda soluzione invece consiste nell'installare sul calcolatore un programma, chiamato *interprete*, che è in grado di eseguire *direttamente* le istruzioni dei programmi scritti nel linguaggio di programmazione di alto livello attuando, per ogni tipo di costrutto del linguaggio interpretato, le azioni appropriate.

La compilazione e l'interpretazione presentano entrambe vantaggi e svantaggi e, per questo motivo, non è possibile stabilire a priori quale delle 2 soluzioni sia da preferire. Di solito, la compilazione consente di ottenere programmi che vengono eseguiti *più velocemente* rispetto all'interpretazione perché non richiedono l'intermediazione dell'interprete. L'approccio interpretato invece ha come vantaggio fondamentale una maggiore semplicità e velocità nella scrittura dei programmi. E' possibile infatti per il programmatore provare *incrementalmente* il programma *durante* la sua scrittura in modo da verificare immediatamente il comportamento di ciascuna istruzione.

MATLAB è un esempio di linguaggio interpretato. Per questo motivo per esempio in MATLAB non occorre dichiarare le variabili. Esse risultano definite nel punto in cui vengono utilizzate per la prima volta. Inoltre, il loro tipo è *dinamico*, nel senso che esso

può cambiare durante l'esecuzione del programma per effetto di assegnamento di valori che appartengono a tipi diversi.

1.2 Ambienti di MATLAB

L'interfaccia principale di MATLAB è composta da diverse finestre che è possibile affiancare, spostare, ridurre a icona, ridimensionare e così via. Le finestre più usate sono *Command Window*, *Workspace*, *Current Folder*, *Command History* e la finestra dell'*Editor*.

1.2.1 Command Window

In questa finestra è possibile inserire comandi che verranno immediatamente eseguiti non appena si clicca invio. Inoltre è in questa finestra che appaiono gli output dei programmi che lanciamo.

Per usare questa finestra come *prompt* dei comandi (cioè per navigare tra le cartelle, vedere il loro contenuto ecc..) si possono digitare i seguenti comandi e poi cliccare invio:

pwd Per conoscere la cartella attuale in cui ci si trova.

cd Per muoversi tra cartelle: seguito da '.' torna alla cartella al livello precedente; per far completare automaticamente il nome della cartella di destinazione si può schiacciare il tasto TAB.

ls Per vedere il contenuto della cartella.

clc Pulisce la Command Window, cioè non si vedrà più il testo digitato, però tutto ciò che è stato fatto rimane comunque in memoria.

La Command Window si può usare anche come semplice calcolatrice: si possono fare le normali operazioni tra numeri (somma +, differenza −, moltiplicazione ∗, divisione /, elevamento a potenza^e parentesi ()), oppure si possono assegnare valori a delle variabili e usare i nomi delle variabili nei calcoli. Se alla fine di una riga di comando si mette il punto e virgola ';' si sopprime l'output, però il comando viene ugualmente eseguito.

La Command Window è molto comoda perché si vede immediatamente l'effetto di ogni comando, però non è possibile tenere una traccia ordinata del codice che si sta scrivendo. Per questo invece di scrivere direttamente qui, in generale, scriveremo in un file a parte, attraverso l'editor di testo di Matlab. Prima di passare all'editor vediamo rapidamente le altre interfacce di MATLAB.

Per vedere quali variabili sono attualmente in uso si può usare il comando `who` oppure, per avere informazioni più dettagliate si può usare `whos`. Le stesse informazioni si trovano anche nello *Workspace*.

Inoltre è possibile utilizzare i comandi `lookfor`, `help` e `doc` per avere informazioni sulle varie funzioni implementate in MATLAB.

Workspace

Finestra nella quale si possono vedere tutte le variabili memorizzate. Con un doppio click sopra una di esse si apre l'array editor (tipo foglio di Excel) dove si può leggere (e anche modificare) il contenuto delle variabili.

Current Folder

E' la prima cartella in cui MATLAB cerca il file da eseguire. Ci viene mostrato tutto il suo contenuto.

Commnd History

Rimane traccia di tutti i comandi digitati nella Command Window, in ordine cronologico.

1.2.2 Editor, creare un nuovo SCRIPT

E' la finestra nella quale scriviamo i nostri codici. Ci permette di realizzare diversi tipi di file. Oggi ci occupiamo degli **script**. Per aprirne uno nuovo: andare su `NUOVO > SCRIPT` (in alto a sinistra, la sequenza esatta dipende dalla versione di MATLAB utilizzata).

Tutto ciò che scriviamo in uno SCRIPT si potrebbe anche scrivere direttamente nella Command Window, ma scrivendolo qui possiamo organizzarlo meglio. E' il tipo più semplice di file MATLAB che possiamo generare perché non contiene nè variabili di ingresso nè di uscita, e per esempio ci serve per automatizzare azioni che MATLAB dovrà eseguire più volte

Per scrivere dei **commenti** basta far precedere ogni riga di commento dal simbolo '%'.

Ogni file può essere salvato in qualsiasi cartella. L'estensione dei file MATLAB è `.m`. E' importante scegliere bene il **nome del file**. In particolare i nomi dei file di MATLAB:

- Devono necessariamente cominciare con un carattere alfabetico (non possono cominciare con un numero).
- Possono contenere lettere e numeri (esempio Esercizio1.m, Es2Lezione1.m); MATLAB è *case-sensitive* cioè distingue tra lettere minuscole e maiuscole.
- Non possono contenere segni di punteggiatura, a parte l'underscore '_' (in particolare non si può usare il normale trattino '-').
- Sarebbe meglio evitare di sovrascrivere funzioni MATLAB già esistenti (per esempio è meglio non chiamare un proprio file `cos.m` perché esso già rappresenta un comando di MATLAB).

Quando **si comincia un nuovo file** è bene scrivere un commento chiaro che descriva il suo scopo. Inoltre è comodo poter pulire lo schermo, cancellare i valori precedentemente assegnati alle variabili, e chiudere le eventuali finestre aperte (per esempio quelle dei grafici): a tale scopo si possono usare rispettivamente i comandi `clc`, `clear all`, e `close all`.

Per **ottenere informazioni sui comandi**: se si conosce già il nome del comando e si vuole avere qualche dettaglio in più su come funziona si può digitare `help` seguito dal nome del comando. In questo modo si ottengono delle informazioni sintetiche (spesso sufficienti); per informazioni ancora più dettagliate si può digitare `doc` seguito dal nome del comando (in questa parte di documentazione sono spesso presenti anche degli esempi).

Se invece non si conosce il nome specifico del comando si può usare `lookfor` seguito da una parola chiave (Es: `lookfor cosine`).

Per quanto riguarda le **variabili** $MATLAB$ non richiede alcun tipo di dichiarazione o dimensionamento. A differenza dei normali linguaggi di programmazione (C, Pascal, Fortran) non occorre dichiarare le variabili. L'assegnazione coincide con la dichiarazione.

Esercizio .1 (Primi comandi MATLAB, svolto a lezione)**.** Creare un nuovo `script` MATLAB. Scrivere al suo interno qualche commento riguardante i nomi che si possono dare ai file. Salvare questo file con il nome di `Esercizio1_PrimiComandi.m`. Provare i comandi nominati in classe (`pi, exp, 1i, format, lookfor, help, doc`) le funzioni goniometriche ed esponenziali.

1.3 Vettori e matrici

Per creare un **vettore** esistono diverse possibilità:

- Mediante inserimento diretto di ogni sua componente:

```
a = [3, 12, 25, 4];   % se gli elementi del vettore sono separati da ...
b = [3 12 25 4];      % ... virgole o spazi vuoti -> vettore RIGA
c = [3; 12; 25; 4];   % separando con punti e virgola creo vettori COLONNA
c';   % il comando ' traspone il vettore (da riga a colonna o viceversa)
```

- Come l'elenco ordinato di N nodi equispaziati nell'intervallo $[a, b]$ (estremi inclusi):

```
a = 2; b = 7; N = 250;
x = linspace(a,b,N);
```

- Come l'elenco dei punti che si trovano partendo dall'estremo a di un intervallo, muovendosi con passo p fino a raggiungere l'altro estremo (b) dell'intervallo (il passo può anche essere negativo):

```
a = 2; b = 6; p = 0.5;
x = a:p:b;   % genera [2  2.5  3  3.5  4  4.5  5  5.5  6]
a = 100; b = 95; p = -1;
y = a:p:b;   % genera [100  99  98  97  96  95]
```

- Casi particolari

```
u = ones(4,1)   % vettore con 4 righe e 1 colonna contenente tutti 1
z = zeros(1,7)  % vettore con 1 riga e 7 colonne contenente tutti 0
t = 3*ones(4,1) % vettore con 4 righe e 1 colonna contenente tutti 3
```

Dato un vettore x è poi possibile

- Determinarne la lunghezza `length(x)`;
- Accedere alla sua componente i-esima `x(i)`;
- Modificare la componente i-esima, es. `x(i) = 50`;
- Accedere e/o modificare le componenti dalla i alla h, es:

```
x = 1:1:6;            % genera [1 2 3 4 5 6]
x(3:5) = [55,66,77]; % x diventa [1 2 55 66 77 6]
```

- Accedere all'ultima componente del vettore (senza doverne conoscere la lunghezza) `x(end)`;
- Fare varie operazioni, come *somma*, *sottrazione*, *moltiplicazione divisione* componente per componente (con i comandi $+$, $-$, $.*$ e $./$), o anche il prodotto scalare (cioè dati due vettori riga `a` e `b`, si fa $\texttt{a} * \texttt{b}'$). Le operazioni puntuali, componente per componente, sono molto importanti perché ci permettono di evitare cicli in molte occasioni.

Per creare delle **matrici** si procede o in modo analogo ai vettori (per inserimento diretto, o mediante `ones, zeros`) oppure si possono creare matrici particolari con comandi come `eye, diag`. Su di esse si possono applicare per esempio i seguenti comandi

- `length, size`: restituiscono rispettivamente la dimensione maggiore e entrambe le dimensioni di una matrice.
- `det, inv, eig` per ottenere il determinante, l'inversa, gli autovalori e gli autovettori di una matrice (guardare l'help di MATLAB per maggiori dettagli)

Esercizio .2 (Matrici e Vettori, svolto a lezione)**.** Definire vari vettori e matrici attraverso le modalità viste sopra. Provare ad eseguire alcune operazioni.

Le matrici poi si possono manipolare in vario modo, vediamo qualche esempio attraverso il prossimo esercizio guidato.

Esercizio .3 (Matrici, da svolgere autonomamente 20 min)**.**

a. Creare i vettori $v = [0, \pi, 2\pi]$ e $w = [0, 1, 2]$. Calcolarne la somma, il prodotto scalare. Calcolarne il prodotto e il quoziente componente per componente. Moltiplicare w per 10. Calcolare il seno e il coseno di v.

b. Creare il vettore riga $z1$ contenente tutti 1, di dimensione 1000. Creare il vettore colonna $z2$ contenente tutti i numeri pari interi tra 1 e 2000. Verificarne le dimensioni con gli appositi comandi e eseguire il prodotto scalare.

c. Costruire la matrice $A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 3 & 3 & 3 & 3 \\ 6 & 5 & 4 & 3 \end{bmatrix}$, in 2 modi diversi e verificare che le matrici fatte siano uguali. Usare il comando `size` per trovare la dimensione di A.

```
% Mediante inserimento diretto
A <- [1,2,3,4; 3, ... ];
% A partire da 3 vettori
a <- 1:1:4;   b <- ...;
c <- ...
A2 <- [a;b;c]
% Vediamo se sono uguali (== fa il test di uguaglianza)
A == A2  % 1 significa VERO, 0 significa FALSO
% (il confronto viene fatto elemento per elemento)
dimensioniA = size(A)
```

d. Estrarre la prima colonna di A e salvarla; estrarre le sottomatrici (chiamarle rispettivamente B e C) di dimensione 2×2 e 3×3 (che occupano la parte superiore sinistra della matrice A). Calcolare la dimensione degli elementi estratti.

```
% Selezioniamo la 1' colonna di A e la salviamo nel vettore 'k'
k = A(:,1)  % = A(tutte le righe, colonna 1)
dimensioniK = ... ;
% Selezioniamo la sottomatrice di A (2,2) in alto a sinistra (la chiamiamo B)
B = A(1:2,1:2) % = A(righe da 1 a 2, colonne da 1 a 2)
dimensioniB =  ...;
% Selezioniamo la sottomatrice di A (3,3) in alto a sinistra (la chiamiamo C)
C = ...  % = A(righe da 1 a 3, colonne ... )
dim ...
```

e. Porre l'elemento nella seconda riga, terza colonna di A uguale a 55 e, quello nella prima riga, quarta colonna uguale a 33.

```
A(2,3) = 55;
A ... ;
```

f. Estrarre, e salvare in un vettore, la diagonale di C. Creare poi la matrice D che sia una matrice che ha per diagonale il vettore $j = [1, 2, 3, 4]$ e tutti gli altri elementi nulli.

   Suggerimento: guardare l'help del comando `diag`.

g. Calcolare il determinante, l'inversa, gli autovalori e gli autovettori di D.

h. Creare la matrice identità 5×5.

   Suggerimento: comando `eye`.

1.4 Cicli

In *MATLAB* la ripetizione di blocchi di istruzioni per *un numero di volte specificato e in modo incondizionato* viene eseguita tramite l'istruzione di ciclo **FOR ...END** la cui sintassi è

```
for indice = espressione
corpo del ciclo
end
```

Dove `indice` è una quantità che assume diversi valori a seconda di `espressione`, e `end` segna la fine del blocco di istruzioni da ripetere.
In `espressione` possiamo scrivere sequenze di indici arbitrarie, ad esempio
`for i = [2, 3, 5, 7, 11, 13, 17, 19]`
oppure indicare intervalli di numeri successivi, ad esempio
`for i = 1:100` (se il passo non è indicato, quello di default è 1),
o anche andare in ordine decrescente, ad esempio `for i=100:-1:1`.

Se si ha la necessità di ripetere una o più istruzioni fintanto che una condizione sarà verificata non sapendo a priori il numero di ripetizioni si usa l'istruzione **WHILE ...END**, la cui sintassi è:

```
while condizione
blocco di istruzioni
end
```

Dove blocco di istruzioni verrà eseguito fintanto che condizione risulta vera.

1.4.1 Il calcolo di π

Per il calcolo di π esistono alcuni importanti algoritmi non tutti convergenti per motivi di instabilità. Di seguito diamo cenno di 2 di questi algoritmi.

Esercizio .4 (Algoritmo di Archimede, 15 min)**.** Mediante questo algoritmo, π è approssimato con l'area del poligono regolare di 2^n lati inscritto nella circonferenza di raggio 1 (che ha area uguale a π). Indicando con b_i il numero di lati dell'i-esimo poligono regolare iscritto, $s_i = \sin\left(\frac{\pi}{2^i}\right)$ e A_i la corrispondente area, l'algoritmo si può così descrivere:

$$
\begin{aligned}
&b_1 \leftarrow 2; \quad s_1 \leftarrow 1;\\
&\text{for } i \leftarrow 1 \text{ to } n\\
&\quad A_i \leftarrow b_{i-1}\, s_{i-1};\\
&\quad s_i \leftarrow \sqrt{\frac{1-\sqrt{1-s_{i-1}^2}}{2}}\\
&\quad b_i = 2b_{i-1}\\
&\text{end for}
\end{aligned}
$$

Provare prima con n piccolo (es. $n = 10$). Poi provare con $n = 30$: commentare i problemi che appaiono alla luce delle considerazioni fatte nel capitolo successivo. Può aiutare il calcolo di `s2` $= s^2$ e la stampa di questa tabella

```
1     fprintf('Iter = %2.0f \t s = %2.1e \t s2 = %2.1e \t b = %2.1e \t A = %4.3e \n', ...
2               i, s, s2, b, A);
```

Esercizio .5 ($4arctan(1)$, da svolgere autonomamente, 15 min)**.** $\pi = 4\arctan(1)$. Usando l'espansione di Taylor di arctan(1), π è approssimato con la formula:

$$
arctan(1) = \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7}\right).
$$

Indicando con q_i la somma al passo i, e volendo arrestare l'algoritmo quando la differenza tra due iterate successive è minore di un certo valore `tol`, l'algoritmo si può descrivere come segue

Inizializzare $q_1 \leftarrow 1, i \leftarrow 1$
Inizializzare tol e err
Ciclo WHILE finchè err $>$ tol
    incrementare il contatore i
    $q_i = q_{i-1} + \dfrac{(-1)^{i-1}}{2i-1}$
    calcolare la differenza tra q_i e q_{i-1}
end WHILE
stampare il risultato

Suggerimento: servirsi di una variabile $qOld$ e di un'altra $qNew$ per poterle avere entrambe quando si deve calcolare l'errore, e poi sovrascriverle correttamente.

A coloro che non ricordassero come costruire dei cicli e avessero riscontrato delle difficoltà nello svolgere gli ultimi 2 esercizi, suggerisco di provare ad implementare qualche algoritmo classico.
Come per esempio i seguenti (di sicuro già affrontati nel corso di Programmazione 1): calcolo di m.c.m, M.C.D, algoritmi di ordinamento semplici come InsertionSort, BubbleSort (potete provare a scriverli in un qualsiasi linguaggio di programmazion).

Esercizio .6 (Evitare i cicli for quando possibile, da svolgere autonomamente, 20 min)**.** In $MATLAB$ è possibile evitare la maggior parte dei cicli `for` utilizzando gli strumenti "furbi" che ci sono forniti così da semplificare e velocizzare i codici.

Creare 2 vettori a scelta `a,b` di 10000 elementi ciascuno. Calcolare

$$s = \sum_{i=1}^{10000} a(i) * b(i)$$

sia usando che non usando il ciclo `for`. Confrontare il tempo impiegato.

Suggerimenti:

- Per non usare il ciclo si può utilizzare il comando `sum` applicato ai 2 vettori moltiplicati tra loro puntualmente;
- Per confrontare i tempi di esecuzione si possono usare i comandi `tic` e `toc`. Il comando `tic` avvia il cronometro, e il comando `toc` lo arresta

```
% definizione dei vettori a e b ...
somma1 = 0; somma2 = 0;  % inizializzazione dei vettori contenenti i risultati
tic
% 1' metodo di calcolo
toc
tic
% 2' metodo di calcolo
toc
```

Esercizio .7 (Grafici, autonomamente)**.** Tracciare il grafico della funzione $\sin x$ per $x \in [-\pi, \pi]$. Colorarlo di verde. Mettere la legenda, le etichette agli assi e dare un titolo al grafico.

Suggerimenti: Definire un vettore x di 100 punti tra $-\pi$ e π. Sia ora $y = sin(x)$. Digitare `plot(x,y,'*g')`. MATLAB disegna un punto di colore verde (green) per ogni coppia di coordinate x e y.

Provare anche `plot(x,y,'-m')`. Per altre combinazioni di simboli e colori vedere il `doc` del comando `plot`.

Per legenda titolo ed etichette usare i seguenti comandi: `legend('sin(x)'), xlabel('x'), ylabel('y'), title('Ecco il grafico di sin(x)')`.

Capitolo 2

Analisi degli errori numerici

La rappresentazione, necessariamente finita, dei numeri all'interno di un calcolatore è una delle fonti principali di errori, detti appunto *errori di rappresentazione*.

Infatti, per memorizzare ogni numero è disponibile una quantità limitata di bit: di conseguenza l'insieme dei numeri rappresentabili ha un limite superiore, un limite inferiore e un numero finito di cifre decimali. In particolare i numeri irrazionali come π e quelli periodici come $1/3$ sono sostituiti da una loro approssimazione con un numero finito di cifre decimali. Inoltre la distanza tra un numero e il suo successivo non è infinitamente piccola, ma è un valore ben determinato, che viene chiamato *epsilon macchina*.

Esercizio .8 (Aritmetica finita, svolto in classe)**.**

a. Determinare il più grande numero rappresentabile con $MATLAB$ (e vedere cosa succede se lo si moltiplica per 2, o per 10).

   Suggerimento: usare il comando `realmax`.

b. Determinare qual è il più piccolo numero che sommato ad 1 restituisce un numero maggiore di 1, e che sommato a 1000 restituisce un numero maggiore di 1000. Commentare.

   Suggerimenti:

   - Tale numero può essere individuato attraverso il comando `eps` di $MATLAB$. In particolare `eps`(N) restituisce il più piccolo numero che sommato ad N dà un numero maggiore di N (verificarlo!).
   - Senza utilizzare il comando di $MATLAB$, tale valore può essere determinato attraverso un ciclo `while`:

```
myeps = 1;  % inizializzazione
while 1 + myeps > 1
myeps = myeps/2;
end
myeps = 2*myeps % Perche' nel ciclo ho diviso anche l'ultima volta
```

     Con questi due metodi si ottiene lo stesso risultato? Confrontare i risultati ottenuti attraverso il comando ==.

c. Determinare il più piccolo numero rappresentabile con $MATLAB$.

   Suggerimento: combinare il comando `eps` con `realmin`.

2.1 Cancellazione numerica

A causa dell'aritmetica finita del calcolatore si incorre in alcuni errori numerici: per esempio quando si *cancellano cifre significative* si perde precisione nella rappresentazione del risultato. Tale fenomeno è detto *cancellazione numerica*.

È possibile quantificare gli errori commessi nell'ottenere un valore approssimato $\hat{x}$ rispetto ad un valore esatto x attraverso

$$\text{l'errore assoluto:} \quad |\hat{x} - x|,$$

$$\text{l'errore relativo:} \quad \frac{|\hat{x} - x|}{|x|}.$$

Esercizio .9 (Cancellazione numerica, 15 min)**.** Si calcoli il valore dell'espressione

$$y = \frac{(1 + x) - 1}{x}$$

al variare di x in un opportuno intervallo vicino allo zero. Si osservi il fenomeno della cancellazione numerica rispetto al valore vero (che è ovviamente 1) calcolando per esempio l'errore relativo e facendo dei grafici. Commentare.

Suggerimenti:

- Per esempio si può prendere come x un vettore di 1000 nodi equispaziati nell'intervallo
$$[-1e-10, 1e-10] \equiv [-10^{-10}, 10^{-10}].$$
Provare anche con intervalli diversi.

- Per calcolare l'errore relativo

```
errRel = abs(y - valEsatto)./abs(valEsatto);
```

- La stampa di due figure in *MATLAB* (con titolo ed etichette per gli assi) può essere effettuata mediante le seguenti istruzioni

```
figure;
plot(x,y);
title('Valore dell''espressione')
xlabel('x')
figure;
plot(x,errRel);
title('Errore relativo')
xlabel('x')
```

- I risultati ottenuti dovrebbero essere simili a quelli riportati in Figura 3.4.

Esercizio .10 (Cancellazione numerica 2, svolto in classe)**.** Calcolare la derivata della funzione $f(x) = x^3 + 1$ mediante la formula del rapporto incrementale

$$f'(x) = \frac{f(x + h) - f(x)}{h}.$$

per valori di $h = 10^{-i}$ con $i = 1, 2...16$ ed $x = 1$.

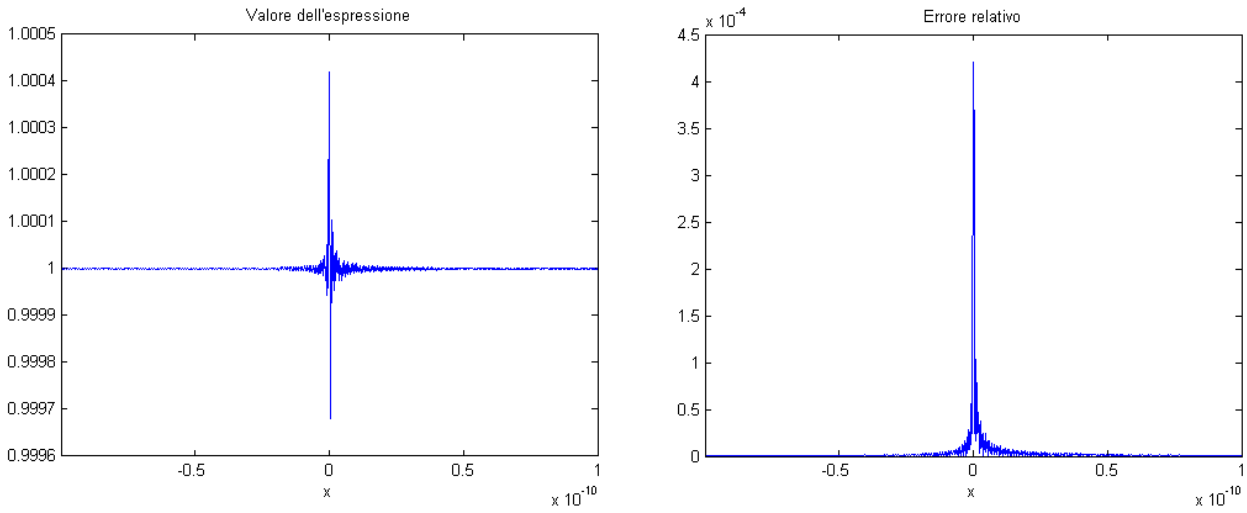

Figura 2.1: Risultati esercizio .9: Grafico della funzione e dell'errore relativo

Confrontare poi i valori ottenuti con il valore esatto (calcolato sapendo che $f'(x) = 3x^2$); calcolare l'errore assoluto, quello relativo e riportare in un grafico l'andamento dell'errore relativo. Commentare i risultati in base al fenomeno della cancellazione numerica.

Suggerimenti:

- Per definire la funzione e i valori di h con cui effettuare i calcoli si può seguire lo pseudocodice

```
x <- 1;
i <- vettore da 0 a 16 con passo 1;
% Ricordare che le operazioni tra vettori,
% che devono essere eseguite elemento per elemento,
% vanno precedute dal punto (es: a.*b, a./b, a.^b, ...)
h <- 10^(-i);
f <- x^3 + 1;  %Definire f come funzione di x (ricordare @(x)...)
df <- (f(x+h)-f(x))/h;
```

- Per poter meglio apprezzare l'andamento dell'errore in funzione di i è consigliabile un **grafico in scala semi-logaritmica**: si ottiene attraverso il comando `semilogy`. In un tale grafico i valori in y non sono equispaziati: infatti è costruito in modo che ci sia la stessa distanza tra ogni potenza di 10, cioè tra $1 = 10^0$ e $10 = 10^1$ c'è la stessa distanza che c'è tra $10 = 10^1$ e $100 = 10^2$, vedi anche Figura 2.2. In questo modo si può vedere meglio la differenza di valori molto piccoli (come quelli che di solito assumono gli errori) che altrimenti sarebbero confinati in un angolo del grafico vicino allo zero.

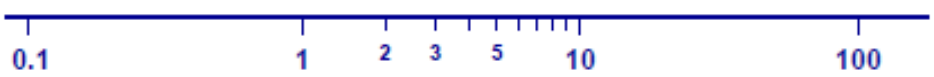

Figura 2.2: Scala logaritmica

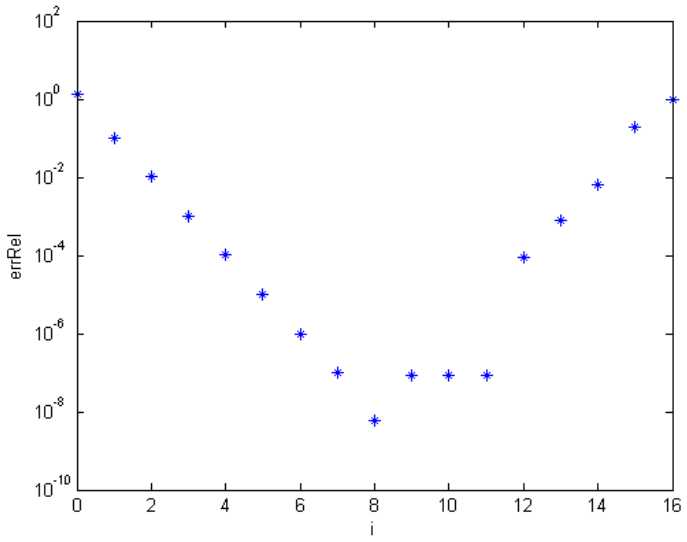

Figura 2.3: Risultati esercizio .10: errore relativo in funzione dell'ordine di grandezza di h.

- Il grafico dell'errore relativo nel calcolo della derivata deve essere simile a quello riportato in Figura 2.3.

Cosa fare per evitare la cancellazione numerica? Una prima risposta è di trovare un'espressione più stabile, ovvero tale da non far aumentare gli errori introdotti dalla formulazione del problema.

Per esempio, volendo valutare l'espressione $\sqrt{x+\delta} - \sqrt{x}$ per $\delta \to 0$ si vede subito che la sottrazione tra valori quasi uguali porterebbe al fenomeno della cancellazione numerica. Si può allora manipolare l'espressione ed ottenerne una equivalente in cui non compaia la sottrazione. In questo caso *razionalizzando* si ottiene la seguente espressione stabile

$$\frac{\delta}{\sqrt{x+\delta}+\sqrt{x}}.$$

Vediamo ora un caso più interessante

Esercizio .11 (Formulazioni stabili e instabili, autonomamente 20 min)**.** Si calcoli il valore dell'espressione

$$y = (x-1)^7 \tag{2.1}$$

per x vicino ad 1. Si noti che usando la formulazione in (2.1) si ottiene un risultato numericamente stabile. Invece se si decide di sviluppare la potenza settima cioè si usa l'espressione

$$z = x^7 - 7.0x^6 + \ldots - 1$$

la formulazione non è più numericamente stabile. Plottare entrambi i risultati.

Suggerimenti:

- Come intervallo scegliere per esempio

```
t = linspace(0.99,1.01,1000);
```

- Per sviluppare $(x-1)^7$ si può usare il toolbox simbolico di $MATLAB$ nel modo seguente

```
Digitare nella command window (non nel file)
   syms x
   expand((x-1)^7)
Copiare dalla command window il risultato ottenuto,
e incollare nel file dell'esercizio:
   z = @(x) ...risultato copiato...
```

  N.B. La variabile scritta all'interno di @( ) è muta.

- Il disegno di due funzioni su una stessa finestra e la presenza della legenda può essere ottenuta con il seguente codice

```
figure;
plot(t,y(t),'b*-');
hold on
plot(t,z(t),'r-.');
title('Valutazione delle due formulazioni del polinomio')
legend('Formulazione stabile','Formulazione instabile')
hold off
```

- Il grafico ottenuto deve essere simile a quello in Figura 2.4.

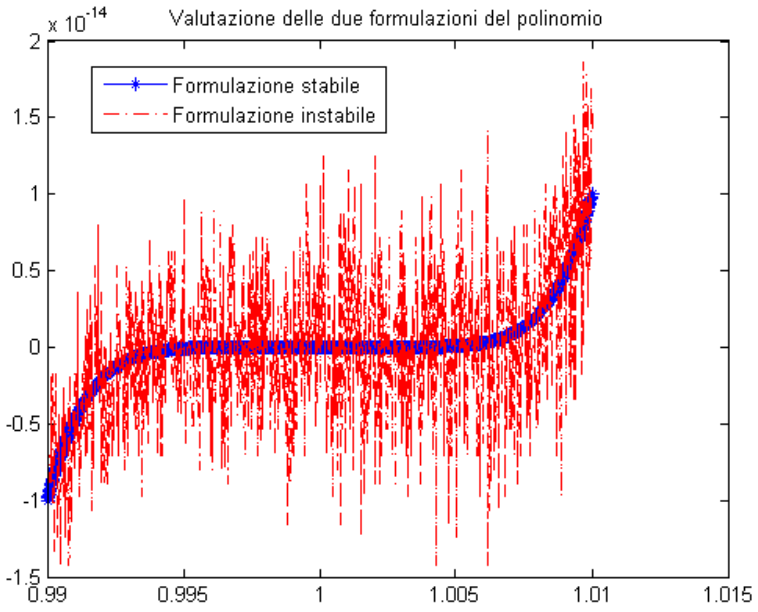

Figura 2.4: Risultati esercizio .11: Formulazione stabile ed instabile del polinomio.

2.2 Stabilità e condizionamento

Dato un problema matematico possiamo distinguere schematicamente, per quanto riguarda la propagazione degli errori, il comportamento del problema e il comportamento di un particolare algoritmo utilizzato per risolvere tale problema.

Nel primo caso, facendo l'ipotesi ideale di essere in grado di risolvere esattamente il problema, si è interessati a vedere come eventuali perturbazioni sui dati del problema si trasmettono sui risultati. Per caratterizzare un problema rispetto a questo tipo di comportamento si utilizza comunemente il termine di *condizionamento*. Più precisamente, si parla di *problema malcondizionato* quando piccole perturbazioni sui dati iniziali (input) influenzano eccessivamente i risultati (output).

Nel caso di un algoritmo, per indicare il suo comportamento rispetto alla propagazione degli errori è più usuale il termine di *stabilità*. Si dirà quindi *algoritmo stabile* un algoritmo nel quale la successione delle operazioni non amplifica eccessivamente gli errori di arrotondamento.

La distinzione tra il condizionamento di un problema e la stabilità di un algoritmo è importante perché, mentre per un problema bencondizionato è possibile, in generale, trovare algoritmi stabili, per un problema malcondizionato può essere opportuna una sua riformulazione.

Vediamo un esempio di un problema malcondizionato.

Esercizio .12 (Problema mal condizionato, autonomamente 20 min)**.** Consideriamo il polinomio

$$p(x) = (x-1)(x-2)(x-3)\dots(x-6) = x^6 - 21x^5 + 175x^4 - 735x^3 + 1624x^2 - 1764x + 720$$

le cui radici sono $1, 2, \dots, 6$. Variamo leggermente il coefficiente di x^2 (da 1624 a 1625): mostrare che le radici cambiano e che non sono più tutte reali! (Attraverso un grafico e/o calcolando le radici)
Provare anche altre perturbazioni. Commentare.

Suggerimenti:

- Per costruire un polinomio conoscendo le sue radici si usa il comando `poly`

```
r = 1:1:6; % vettore contenente le radici del polinomio
p = poly(r);
% POLY restituisce il vettore p contenente i coefficienti del polinomio
% che ha come radici quelle elencate nel vettore r.
% I coefficienti in p sono in ordine decrescente.
```

- Ora costruiamo un polinomio $p1$ il cui coefficiente del termine di secondo grado sia stato variato

```
p1 = p;
p1(5) = 1625; %  Perturbazione nel 5' coefficiente cioe' quello di x^2
```

- Per `valutare` un polinomio con coefficienti memorizzati nel vettore p su tutti i punti di un vettore x si può usare il comando `polyval`. L'intervallo di punti x su cui è interessante valutare i polinomi è quello contenente le radici, quindi per esempio $[0.8, 6.2]$.

```
x = 0.8:0.1:6.2;
y = polyval(p,x);
y1 = ... valutare p1 ...
```

- Plottare, in uno stesso grafico sia y, sia $y1$. Per visualizzare la griglia usare il comando `grid`. Confrontare con la Figura 2.5.
- Per calcolare le radici (approssimate) di un polinomio dati i suoi coefficienti si può usare il comando `roots`.

```
1     disp('Radici del polinomio iniziale')
2     r = roots(p)
3     disp('Radici dei polinomi con coefficiente perturbato')
4     r1 = roots(p1)
```

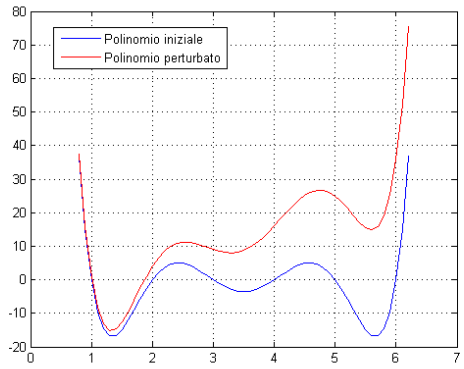

Figura 2.5: Risultato esercizio .12: Grafico del polinomio iniziale e di quello perturbato.

Vediamo un altro esempio di un problema per cui esiste un algoritmo risolvente stabile ed un altro non stabile.

Esercizio .13 (Formulazioni stabili e instabili 2, autonomamente 15 min)**.** Calcolare

$$I_n = e^{-1} \int_0^1 x^n e^x \, dx$$

mediante le seguenti formulazioni.

Formulazione non numericamente stabile. Per $n = 0$ si ha che

$$I_0 = e^{-1} \int_0^1 e^x \, dx = e^{-1}(e^1 - 1).$$

Per $n = 1$, è facile verificare, usando la formula di integrazione per parti, che

$$\int_0^1 x\, e^x \, dx = (x-1)e^x + c$$

quindi $I_1 = e^{-1}$, e più in generale, sempre integrando per parti si ottiene,

$$I_n = e^{-1}(x^{n+1}\, e^x|_0^1 - (n+1)\int_0^1 x^n e^x \, dx = 1 - (n+1)I_n$$

da cui la ricorrenza

$$\boxed{\begin{aligned} I_{n+1} &= 1 - (n+1)I_n \\ I_1 &= e^{-1}\,. \end{aligned}}$$

Formulazione numericamente stabile Dal fatto che l'integranda $x^n e^x > 0$ per $x \in (0, 1]$ si ha che

$$I_n > 0.$$

Inoltre $x \leq 1$ implica $x^2 \leq x$ e più in generale che $x^{n+1} \leq x^n$ da cui $x^{n+1}e^x \leq x^n e^x$ e quindi

$$I_{n+1} \leq I_n.$$

La successione I_n è positiva e non crescente e quindi ammette limite L finito, in particolare si dimostra che questo limite è zero.

Da cui la formula di ricorrenza

$$\boxed{\begin{aligned} I_n &= \frac{1 - I_{n+1}}{n+1} \\ I_{\text{grande}} &= 0. \end{aligned}}$$

Commentare i risultati ottenuti (fare un grafico dove si riportano i valori ottenuti). Capire in particolare quale è la causa dell'amplificazione dell'errore in un caso e del suo smorzamento nell'altro caso.

Suggerimenti:

- Lo pseudocodice che implementa la prima ricorrenza è

```
s <- ... inizializzazione a vettore zero di 100 elementi (usando 'zeros')
s(1) <- exp(-1);        % Caso base
for n <- 1 to 99  do    % Ciclo iterativo
s(n + 1) <- 1 - (n + 1)*s(n);
end
```

- Poiché i valori calcolati con la prima formulazione tendono rapidamente ad infinito in modulo, ed oscillano tra valori negativi e positivi, è consigliabile un grafico in scala logaritmica in y (`semilogy`), ed in particolare del modulo di s.

```
iterazioni = 1:100;
semilogy(iterazioni,abs(s),'r+',iterazioni, ... )
xlabel('...')
ylabel('...')
legend('Formulazione instabile, in modulo', 'Formulazione stabile')
```

I risultati ottenuti dovrebbero essere simili a quelli riportati in Figura 2.6.

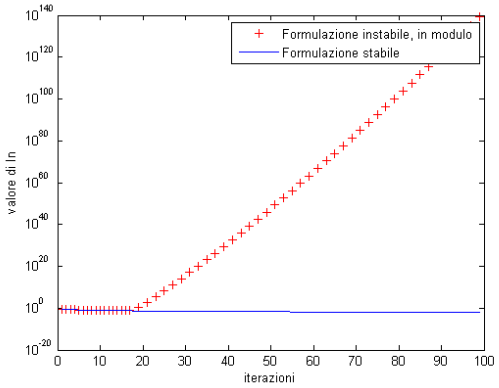

Figura 2.6: Risultati dell'esercizio .13.

Capitolo 3

Ricerca degli zeri di una funzione

3.1 Richiami di MATLAB: come creare le funzioni

In MATLAB oltre ai normali file si possono creare delle funzioni. Esistono sia le funzioni interne, quelle che rispecchiano le normali funzioni definite nella pratica matematica, sia le funzioni esterne, quelle che servono ad implementare algoritmi più complicati.

Le funzioni interne possono essere definite direttamente all'interno di ciascuno script, grazie al comando `@( )`, all'interno del quale vanno scritti gli *input* della funzione, come ad esempio:

```
1 f = @(x,y) sin(x)cos(y)
```

Le funzioni esterne vengono definite in file a parte e successivamente possono essere invocate nello script principale.

I file contenenti delle funzioni iniziano con la parola chiave `function` seguita da

- gli **output**: tra parentesi quadre e separati da virgole;
- il **nome della funzione**: è *obbligatoriamente* anche il nome che deve essere dato al file;
- gli **input**: tra parentesi tonde e separati da virgole.

Cioè

```
1 function  [output1, output2] = nome_funzione (input1, input2, input3)
```

Le funzioni possono avere un numero qualsiasi di input ed output (anche zero) e alcuni degli input possono talvolta essere omessi (usando comandi come `varargin`).

Prima di scrivere il corpo di una funzione è utile fare delle **specifiche dettagliate**, contenenti titolo, breve descrizione del funzionamento, descrizione degli input e degli output, un esempio di come invocare la funzione.

Le funzioni non vanno da sole! Devono essere invocate in un altro file, o nella "Command Window", fornendogli gli input necessari.

Esempio di una semplice funzione con 4 input, 1 output e che produce un grafico (notare che le specifiche sono dettagliate anche se la funzione è semplice!)

```
1 function [y0] = disegnaRetta (m, q, a, b)
2 % Disegna una retta
3 %
4 % DESCRIZIONE: Disegna una retta dati il coefficiente angolare e
```

```
% l'ordinata all'origine su un intervallo dato, e restituisce il
% valore della retta in  x=0
%
% INPUT:
% m = coefficiente angolare
% q = ordinata all'origine
% a = estremo sinistro dell'intervallo
% b = estremo destro dell'intervallo
%
% OUTPUT
% y0 = valore assunto dalla retta per x = 0
%
% Esempio
% y0 = disegnaRetta(2,5,-3,7)

x = linspace(a,b,1000);
y = m*x+q;

plot(x,y)
title('Grafico della retta')
xlabel('x')
ylabel('y=mx+q')
y0 = q;
```

3.2 Richiami di MATLAB: creare una libreria di funzioni

In un file si possono invocare o le funzioni di *MATLAB*, quelle già implementate nel sistema, o delle funzioni costruite da noi. Però per poter utilizzare le funzioni implementate da noi, esse devono essere salvate in una posizione visibile al file che le invoca.

La scelta più semplice è quella di salvare le funzioni nella stessa cartella in cui si salva il file chiamante. Ma quando una funzione deve essere invocata molte volte, in situazioni differenti, non è la scelta migliore.

Allora si possono salvare le funzioni in una cartella a parte, da chiamare per esempio `Libreria`, e aggiungere questa cartella al `path`, cioè all'elenco delle cartelle in cui *MATLAB* cercherà la funzione invocata. Per aggiungere una cartella al `path` ci sono 2 metodi:

- Ogni volta che si apre *MATLAB* si scrive nella `Command Window` (la zona calcolatrice) il comando `addpath` e tra parentesi tonde ed apici il percorso della cartella, ad esempio

```
addpath('home\accounts ... \Scrivania\Calcolo Numerico\Libreria')
```

- Nei menu di *MATLAB* si cerca `Set Path` e si imposta seguendo il percorso guidato: il percorso aggiunto sarà ricordato anche alle aperture successive di *MATLAB* (sui computer del laboratorio potrebbe non essere consentita questa modalità).

3.3 Il metodo di bisezione

Il metodo di bisezione è uno dei metodi per risolvere equazioni non lineari più noti per la sua semplicità ed intrinseca robustezza, ed è basato sul teorema

Teorema 3.1 (Teorema degli zeri)**.** Sia f una funzione continua nell'intervallo $[a, b]$, se $f(a) \cdot f(b) < 0$, allora esiste (almeno) un punto $\xi \in]a, b[$ tale che $f(\xi) = 0$.

Il metodo converge sempre alla radice di $f(x)$ contenuta in $[a, b]$ (se tale intervallo ne contiene una sola) oppure ad una delle radici contenute in $[a, b]$ (se in tale intervallo ne sono presenti più di una). Per garantire la convergenza del metodo allo zero cercato bisogna assicurarsi che esso sia l'unico zero della funzione in tale intervallo (e se richiesto giustificare analiticamente la sua unicità nell'intervallo).

Il metodo di bisezione appartiene alla famiglia dei metodi che '*non possono fallire*' la cui convergenza alla radice è sempre garantita *se sono soddisfatte le condizioni iniziali*: continuità della funzione e differenza di segno agli estremi dell'intervallo. (Per zeri doppi il metodo non funziona, ma questo è ovvio visto che nel caso di uno zero doppio la funzione non cambia di segno nell'intervallo che lo contiene.)

La convergenza del metodo è molto lenta e questo ne costituisce un limite: ad ogni passo si riduce l'errore di 1/2, per ridurlo di 1/10 (1 cifra decimale) occorrono circa $3,3$ passi. Una spiegazione può essere ricercata nel fatto che il metodo non tiene conto del reale andamento della funzione f nell'intervallo $[a, b]$ ma soltanto dei segni. Una caratteristica molto importante che rende questo metodo tuttora valido specialmente nel calcolo automatico è però la sua robustezza.

Esercizio .14 (Metodo di bisezione, commentato insieme, da finire autonomamente 15 min)**.** Costruire una `function` in $MATLAB$ che implementi il metodo di bisezione, cioè una *routine* tale che

- Voglia in **input**: la funzione di cui calcolare gli zeri, gli estremi dell'intervallo in cui cercare gli zeri, la precisione desiderata, e il numero massimo di iterazioni consentite;
- Restituisca in **output**: lo zero trovato, il valore della funzione in tale zero, e un vettore contenente gli zeri approssimati trovati ad ogni iterazione del metodo.

Utilizzare poi questa routine per il calcolo di uno zero di ciascuna delle seguenti funzioni (scegliere bene l'intervallo in cui cercare, in modo che contenga un solo zero e che rispetti le ipotesi del teorema 3.1)

1. $f(x) = x^2 - 2$
2. $f(x) = e^{-2x} + x - 1$
3. $f(x) = x^2 - 2 - log(x)$.

Riportare l'andamento dell'errore relativo.

Suggerimenti:

- Pseudocodice della routine di bisezione

```
function [x,fx,xhist] = bisezione(fname,a,b,tol,maxit)
% Metodo di bisezione
%
% DESCRIZIONE:
% Implementa il metodo della bisezione. Si richiede che f(a)*f(b)<0.
% INPUT:
% fname  :  nome della funzione
```

```
8       % a      :  estremo di sinistra
9       % b      :  estremo di destra
10      % tol    :  tolleranza di arresto
11      % maxit  :  numero massimo di iterazioni
12      % OUTPUT:
13      % x      :  zero trovato
14      % fx     :  valore della f nel punto x
15      % xhist  :  successione delle x trovate
16      %
17      % ESEMPIO:
18      % fname = @(x) x.^2 - 2;
19      % a = 0; b = 2;
20      % tol = 1e-8;    maxit = 100;
21      % [x,fx,xhist] = bisezione(fname,a,b,tol,maxit);
22      % valEsatto = sqrt(2);
23      % semilogy((abs(xhist - valEsatto))/abs(valEsatto),'*');
24      % title('Errore relativo nel calcolo di uno zero di f(x) = x^2-2')
25
26      % Inizializzazione a valori di default dell'output
27      x <- NaN;              % Not a Number
28      fx <- NaN;             % Not a Number
29      xhist <- [];           % Vuoto
30
31      % Verifica degli estremi dell'intervallo
32      if a>=b
33      error('Metodo di bisezione non applicabile (a >= b).');
34      end
35
36      % Valutazione della funzione negli estermi con feval
37      % feval: is usually used inside functions which take function
38      %        handles or function strings as arguments.
39      % Cioe' e' utile perche' cosi' possiamo passare come input proprio il nome
40      % della funzione
41      fa <- feval(fname,a);
42      fb <- ...;
43      % Verifica degli estremi della funzione
44      if sign(fa)*sign(fb) > 0
45      error('Metodo di bisezione non applicabile (fa*fb>0).');
46      end
47
48      % Inizializzazione
49      iter = 0;
50      xhist = repmat(NaN,maxit,1); % repmat: ripete il valore NaN su
51      % maxit righe e 1 colonna
52
53      % Ciclo principale
54      while abs(a-b) > tol && iter<=maxit
55          % Aggiorno contatore delle iterazioni
56          iter <- iter + 1;
57          % Calcolo del punto medio
58          xm <- a+(b-a)/2;
59          % Valuto la funzione nel punto medio
60          fm <- ... ;
61          % Aggiorno gli estremi dell'intervallo in base alla funzione
62          if sign(fa)*sign(fm)<=0    % zero tra [a,xm]
63              b <- xm;
64              fb <- fm;
65          else                       % zero tra [xm,b].
66              a <- ...;
67              fa <- ...;
```

```
68          end
69          % Salvataggio del punto medio per info
70          xhist(iter) <- xm;
71      end
72
73      % Calcolo del punto finale
74      x <- a+(b-a)/2;
75      fx <- fname(x);
76      xhist <- xhist(1:iter); % ridimensiona il vettore (per mandarne in
77      % output solo la parte interessante)
```

- Per l'invocazione della routine di bisezione guardare l'esempio scritto nella specifica.
- Per individuare l'intervallo in cui è sicuramente contenuto uno zero della funzione plottarla.
- Per il calcolo dell'`errore relativo` bisogna fare il confronto con un valore di riferimento: scegliere quello esatto quando lo si conosce; quando invece non lo si conosce scegliere il valore ottenuto con la funzione di $MATLAB$

  `fzero(nome_funzione, x0)`, dove x_0 è un punto sufficientemente vicino allo zero cercato. Nel caso in cui il valore esatto fosse zero, calcolare l'errore assoluto.
- L'andamento dell'errore relativo per il calcolo dello zero positivo della terza funzione è riportato in Figura 3.1.

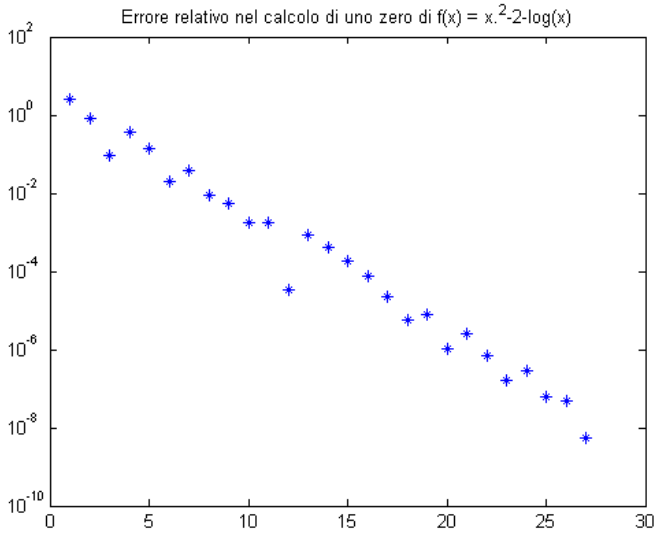

Figura 3.1: Esercizio .14, andamento dell'errore relativo per la terza funzione.

3.3.1 Errore assoluto ed errore relativo

Si consideri una successione $\{x(k)\}_k$ convergente ad x e una stima dell'errore assoluto $|x - x(k)| = e^{(k)}$. Nel caso si usi una tolleranza per l'errore assoluto tol_a, il criterio d'arresto sarà basato sulla disuguaglianza

$$e(k) \leq tol_a$$

mentre, nel caso di una tolleranza per l'errore relativo tol_r, sulla disuguaglianza

$$e(k) \leq tol_r|x(k)|.$$

Se $x = 0$, non è sensato usare una tolleranza per l'errore relativo. Si può usare allora un criterio d'*arresto misto*

$$e(k) \leq tol_r|x(k)| + tol_a \tag{3.1}$$

che diventa, in pratica, un criterio d'arresto per l'errore assoluto quando $x(k)$ è sufficientemente piccolo. Riflettere sui valori appropriati per tol_r e tol_a.

Esercizio .15 (Criterio d'arresto misto, autonomamente 15 min)**.** Modificare la routine di bisezione implementata nell'esercizio .14 in modo che accetti in input sia un parametro di tolleranza assoluta sia uno per la tolleranza relativa. Modificare poi appropriatamente il test d'arresto del ciclo while come suggerito dal criterio d'arresto misto in (3.1). Testare la nuova routine su alcune delle funzioni dell'esercizio .14.

Esercizio .16 (Bisezione su equazione di stato dei gas, autonomamente)**.** L'equazione di stato di un gas reale che lega temperatura, pressione e volume può essere modellizzata mediante l'equazione di Beattie-Bridgeman (estensione della legge ideale dei gas):

$$P = \frac{RT}{V} + \frac{\beta}{V^2} + \frac{\gamma}{V^3} + \frac{\delta}{V^4}$$

ovvero in termini di V come

$$f(V) = \left(RT + \frac{\beta}{V} + \frac{\gamma}{V^2} + \frac{\delta}{V^3}\right)\frac{1}{P} - V$$

dove P è la pressione (atm), V è il volume molare (litro), T è la temperatura (K). Questi sono parametri caratteristici del gas (in generale dipendenti dalla temperatura) ed R è la costante universale dei gas (R = 0,08206). I parametri β, γ, δ sono definiti da:

$$\beta = RTB_0 - A_0 - \frac{Rc}{T^2} \qquad \delta = \frac{RB_0bc}{T^2}$$
$$\gamma = -RTB_0b + aA_0 - \frac{RcB_0}{T^2}$$

con A_0, B_0, a, b, c costanti. Nel caso del gas isobutano si ha

$$A_0 = 16,6037 \quad B_0 = 0,2354 \quad a = 0,11171 \quad b = 0,07697 \quad c = 3,0 \cdot 10^6.$$

Calcolare il valore di V per il quale $f(V) = 0$, fissando $T = 408\,\mathrm{K}$ e $P = 36$ atm. Fare un grafico di $f(V)$.

Suggerimenti:

- Visto che la funzione è complessa, invece di definirla internamente (con `@()`), definirla in una `funtion` a parte del tipo

```
function fV = gas(V)
```

  in cui siano inizializzati tutti i parametri.

- Usare la routine bisezione, definita durante l'Esercizio 4 per il calcolo dello zero

```
[ ... ] = bisezione('gas',Va,Vb,tol,maxit)
```

- Il grafico della funzione dovrebbe essere simile a quello riportato in Figura 3.2. E lo zero della funzione dovrebbe essere $V \simeq 0.2334$.

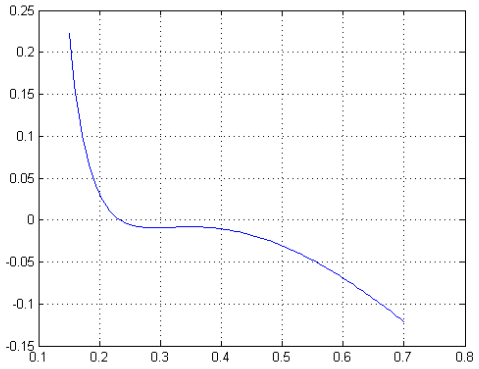

Figura 3.2: Risultati esercizio .16: grafico della funzione

3.4 Metodo del punto fisso

Oltre al metodo di bisezione, esistono altri metodi per cercare gli zeri di una funzione: il metodo del punto fisso, il metodo delle secanti, il metodo delle tangenti e le loro diverse varianti. Il metodo del punto fisso prevede di cercare lo zero di una funzione f attraverso la ricerca del punto fisso di un'altra funzione, chiamiamola g, ottenuta manipolando f. Partendo da una stessa funzione f si possono derivare infinite funzioni g che sono la riscrittura di f nella forma di punto fisso, ma non è detto che ognuna di queste funzioni renda il metodo convergente.

Se l'equazione $f(x) = 0$ è scritta nella forma $x = g(x)$ con $g(x)$ derivabile in un intorno $I(\xi)$ e tale che $\xi = g(\xi)$ se e sole se $f(\xi) = 0$, è possibile utilizzare un metodo iterativo di punto fisso della seguente forma

$$\begin{aligned} x_0 &= \text{approssimazione iniziale nota} \\ x_{k+1} &= g(x_k) \quad k = 0, 1 \dots \end{aligned}$$

per costruire una successione $\{x_n\}$ che approssimi (se converge) lo zero ξ. Esistono infiniti modi di scrivere $f(x) = 0$ nella forma $x = g(x)$ e non tutte le formulazioni portano a convergenza. La convergenza del metodo dipende dalla derivata di $g(x)$ nel punto fisso, più precisamente:

- se $|g'(\xi)| < 1$ il metodo converge,
- se $|g'(\xi)| > 1$ il metodo non converge.

Il codice che implementa una routine per il metodo del punto fisso è il seguente

```
1  function [x,xhist,flag] = puntofisso(gname,x0,tol,maxit)
2  % Iterazione di punto fisso
3  %
4  % DESCRIZIONE:
5  % Implementazione dell'iterazione di punto fisso.
6  %
7  % SINTASSI:
8  % [x,xhist,flag] = puntofisso(fname,x0,tol,maxit)
9  %
10 % INPUT:
```

```
% gname: nome della funzione di punto fisso
% x0:    approssimazione iniziale
% tol:   tolleranza di arresto
% maxit: numero massimo di iterazioni
%
% OUTPUT:
% x:     zero trovato
% xhist: successione delle x trovate
% flag:  se 0 il metodo termina in meno di maxit iterazioni,
%        se 1 il metodo non converge in meno di maxit iterazioni
%
% ESEMPIO:
% gname = @(x) (4 + 11*x - x.^4)/11','x');
% x0 = 1;
% tol = 1e-8; maxit = 100;
% [x,xhist,flag] = puntofisso(gname,x0,tol,maxit);
% semilogy(abs((xhist - sqrt(2))/sqrt(2)));

% Inizializzazione a valori di default dell'output
x = NaN;              % Not a Number
xhist = [];           % Vuoto
flag = 1;             % Non convergenza

% Inizializzazione
iter = 0;
xhist = repmat(NaN,maxit,1);
scarto = tol+1;

% approssimazione iniziale come primo punto della successione
xold = x0;
xhist(1) = xold;

% Ciclo principale (scegliere il criterio d'arresto preferito)
while  ... criterio d'arresto ...
    % Aggiorno contatore delle iterazioni
    iter = ...
    % Calcolo del nuovo punto
    xnew = feval( ... ,xold);
    % Calcolo scarto
    scarto = abs(xnew-xold);
    % Aggiorno punto vecchio come punto nuovo
    xold =  ...
    % Salvataggio dati
    xhist(iter+1) = xnew;
end
if(iter<=maxit)
    flag = 0;
end

% Calcolo del valore finale
x = xnew;
xhist = xhist(1:iter);  % commentare!
```

Esercizio .17 (Punto fisso, commentato insieme, da finire autonomamente 15 min)**.** Si consideri l'equazione

$$f(x) = x - x^{1/3} - 2.$$

a. Mostrare che lo zero esiste ed è unico nell'intervallo [3,5].

b. Utilizzare le seguenti 3 iterazioni di punto fisso (dopo aver verificato che si ottengono davvero manipolando $f(x)$) per determinare lo zero della funzione, utilizzando come punto iniziale $x_0 = 3$:

   (1) $g_1(x) = x^{1/3} + 2$,
   (2) $g_2(x) = (x-2)^3$,
   (3) $g_3(x) = \frac{6+2x^{1/3}}{3-x^{-2/3}}$.

c. Riportare l'andamento dell'errore relativo rispetto al valore trovato utilizzando il comando $MATLAB$ `fzero`.

d. Dare una giustificazione della mancata convergenza dell'iterazione di punto fisso collegata a g_2 e della convergenza negli altri 2 casi.

Suggerimenti:

- Per l'implementazione della routine di punto fisso si veda il codice riportato nelle pagine precedenti, e per la sua invocazione si veda l'esempio riportato nelle specifiche.
- Per far apparire più grafici in un'unica finestra si può usare il comando `subplot(a,b,c)`, dove `a` indica il numero di righe in cui si vuole suddividere la finestra, `b` il numero di colonne, e `c` la posizione che il grafico scritto nella riga sotto il comando dovrà occupare, come nell'esempio:

```
subplot(1,3,1)
semilogy(abs((x1hist-xRif)/xRif),'r+-');
title('Andamento dell''errore relativo per g1')
xlabel('iter')
ylabel('err Relativo')
subplot(1,3,2)
semilogy(abs((x2hist-xRif)/xRif),'*-');
title('Andamento dell''errore relativo per g2')
subplot(1,3,3)
semilogy(abs((x3hist-xRif)/xRif),'go-');
title('Andamento dell''errore relativo per g3')
```

- Il grafico dell'errore dovrebbe essere come quello riportato in Figura 3.3.
- Per i commenti riguardanti la convergenza si ricordi la condizione sulla derivata di $g(x)$.

Esercizio .18 (Bisezione VS Punto Fisso, autonomamente)**.** Si utilizzino sia il metodo di bisezione, sia il metodo del punto fisso per calcolare lo zero della funzione

$$f(x) = x^3 + 4xcos(x) - 2$$

nell'intervallo $I = [0, 2]$. Come iterazione di punto fisso si consideri

$$g(x) = \frac{2 - x^3}{4cos(x)}.$$

Si riporti l'andamento dell'errore relativo rispetto ad una soluzione di riferimento calcolata utilizzando il comando `fzero`.

I risultati ottenuti dovrebbero essere simili a quelli riportati in figura 3.4.

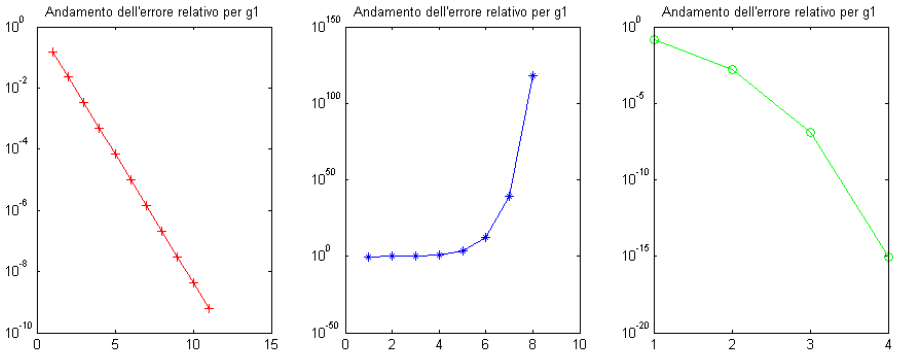

Figura 3.3: Risultati esercizio .17

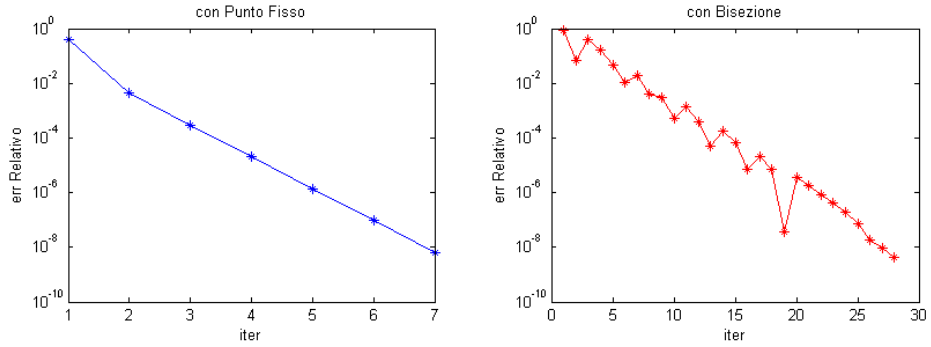

Figura 3.4: Risultati esercizio .18

3.5 Metodo di Newton, o metodo delle tangenti

Il metodo di Newton (detto anche metodo delle tangenti) è sostanzialmente un metodo di *punto fisso*, infatti viene generata una successione di valori che approssimino la soluzione esatta nel modo seguente:

$$x_0 = \text{approssimazione iniziale nota}$$
$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \quad n = 0, 1 \ldots$$

dove quindi la funzione di punto fisso è $g(x) = x - \frac{f(x)}{f'(x)}$.

Esercizio .19 (Newton)**.** Implementare una funzione per il metodo di Newton ed utilizzarla per calcolare gli zeri della funzione $f(x) = x^3 + 4xcos(x) - 2$ nell'intervallo $I = [0, 2]$. Riportare l'andamento dell'errore relativo.

Suggerimenti: per implementare la `function` di Newton si parta dall'iterazione di punto fisso e si eseguano i seguenti passi:

- Copiare e rinominare la routine di punto fisso in Newton.
- Modificare l'intestazione della funzione in modo tale che la funzione (di Newton) abbia in ingresso non direttamente la funzione di punto fisso $g(x)$ ma la funzione di cui trovare lo zero $f(x)$ e la sua derivata $f'(x)$.

- Prima del ciclo inserire una riga di codice nel quale si costruisce la funzione $g(x)$ per l'iterazione di punto fisso del metodo di newton, e modificare di conseguenza la riga, nel ciclo, in cui tale funzione viene utilizzata.
- Aggiornare i commenti.
- Testare il metodo scritto sulla funzione dell'esercizio.

L'andamento dell'errore dovrebbe essere come riportato in Figura 3.5.

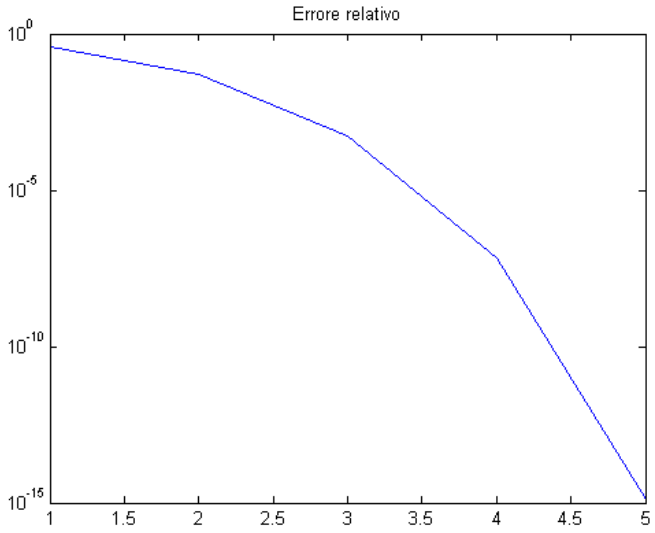

Figura 3.5: Risultati esercizio .19

3.5.1 Il metodo di Newton potrebbe non convergere

Esercizio .20 (Newton, analisi di convergenza)**.** Si utilizzi il metodo di Newton per il calcolo della radice quadrata x di un numero positivo A, dopo aver osservato che $x^2 = A$. Si faccia partire l'algoritmo da un punto iniziale a scelta e si riprovi poi con il suo opposto, e poi con $x = 0$. Commentare i risultati alla luce della convergenza o meno del metodo e del valore a cui converge.

3.6 Ordine di convergenza di un metodo

Definizione 3.1. Sia $\{x_k\}$ una successione convergente ad α. Consideriamo l'errore assoluto in modulo al passo k, $|e_k| = |x_k - \alpha|$. Se esiste un reale $p \geq 1$ e una costante reale positiva $\gamma < +\infty$ t.c.

$$\lim_{k\to\infty} \frac{|e_{k+1}|}{|e_k|^p} = \lim_{k\to\infty} \frac{|x_{k+1} - \alpha|}{|x_k - \alpha|^p} = \gamma,$$

allora la successione $\{x_k\}$ ha ordine di convergenza p.

Se $p = 1$ parleremo di convergenza *lineare*; se $p = 2$ di convergenza *quadratica*; se $p = 3$ di convergenza *cubica* e così via.

Per calcolare l'ordine di convergenza di un metodo si può usare la precedente definizione, ma si può anche derivare un modo alternativo e più rapido.

Considerando che

$$\lim_{k\to+\infty} \frac{|x_{k+1}-\alpha|}{|x_k-\alpha|^p} = \gamma > 0,$$

si ricava

$$p = \lim_{k\to\infty} \frac{\log|x_{k+1}-\alpha| - \log\gamma}{\log|x_k-\alpha|} \approx \lim_{k\to\infty} \frac{\log|x_{k+1}-\alpha|}{\log|x_k-\alpha|}. \tag{3.2}$$

Naturalmente **se**, come spesso accade, **non si conosce la radice** α, si può considerare in (3.2) invece di α una sua approssimazione x_m con m più grande di $k+1$.

Esercizio .21 (Ordine di convergenza, insieme)**.** Si consideri la seguente iterazione di punto fisso

$$x_{n+1} = x_n + e^{(1-x_n)} - 1.$$

a) Si ricavi la funzione $f(x)$ che ha per zero l'unico punto fisso di questa iterazione. Si mostri che lo zero è unico.

b) Si provi **teoricamente** la convergenza di tale iterazione all'unico punto fisso.

c) Provare anche **numericamente** la convergenza di tale iterazione: cioè riportare l'andamento dell'errore relativo e determinare l'ordine di convergenza del metodo di punto fisso (facendo riferimento alla formula in (3.2)).

d) Cercare lo zero di $f(x)$ usando sia il metodo di bisezione, sia il metodo di Newton; per entrambi riportare l'andamento dell'errore relativo e determinare l'ordine di convergenza del metodo.

Suggerimento: Si dovrebbe trovare che per bisezione si ha convergenza lineare, mentre per questa iterazione di punto fisso e per newton convergenza quadratica.

Esercizio .22 (Ordine di convergenza 2, autonomamente)**.** Si considerino le funzioni

$$f_1(x) = \log\left(\frac{2}{3-x}\right), \quad f_2(x) = x^3 - 3.$$

a) Riportare in uno stesso grafico l'andamento delle due funzioni in un intervallo che contenga la loro intersezione; mettere titolo e legenda al grafico.

b) Mediante il metodo di Newton determinare l'unica intersezione $x*$ di $f_1(x) = f_2(x)$

c) Calcolare anche il numero di iterazioni e l'ordine di convergenza (rispetta le attese?).

d) Se si commette un piccolo errore nella formula della derivata (un cambio di segno o si altera di poco un numero), cosa succede?

Suggerimento: Il metodo di Newton ha normalmente ordine di convergenza 2. Quando l'ordine non viene raggiunto potrebbe essere per una delle situazioni descritte nella prossima sezione (molteplicità degli zeri); ma una causa più banale potrebbe essere aver commesso un errore nel calcolo della derivata. In tal caso infatti, a volte, il metodo converge lo stesso e l'unico segnale che indica la presenza dell'errore è l'ordine di convergenza < 2.

Il fatto che il metodo di Newton converga anche se la derivata non è esatta è alla base di alcune modifiche di tale metodo che diminuiscono il costo computazionale dell'algoritmo. Noi non le vediamo.

3.7 Il metodo di Newton - approfondimenti

3.7.1 Zeri multipli

Nel caso di zeri con molteplicità maggiore di 1 il metodo di Newton è *malcondizionato*. Inoltre il suo ordine di convergenza diminuisce.

Questi due concetti sono distinti.

Il fatto che l'ordine di convergenza diminuisca si vede facilmente studiando la derivata della funzione di punto fisso. Ricordiamo che la funzione di punto fisso legata all'iterazione di Newton è

$$g(x) = x - \frac{f(x)}{f'(x)}. \tag{3.3}$$

Supponiamo che f abbia come radice 0 con molteplicità m allora esiste una funzione h, $h(0) \neq 0$, tale che f possa essere riscritta come

$$f(x) = x^m h(x) \tag{3.4}$$

la cui derivata è

$$f'(x) = mx^{m-1}h(x) + x^m h'(x). \tag{3.5}$$

Quindi sostituendo (3.4) e (3.5) in (3.3) otteniamo

$$g(x) = x - \frac{x^m h(x)}{x^{m-1}(mh(x) + h'(x))}, \tag{3.6}$$

e derivando (dopo un po' di calcoli) otteniamo

$$g'(0) = 1 - \frac{1}{m}. \tag{3.7}$$

É chiaro che più m è grande, più $|g'(0)|$ è vicino ad 1: ciò implica una convergenza sempre più lenta.

Invece il fatto che il metodo sia mal condizionato implica che, anche permettendogli di fare moltissime iterazioni e anche imponendogli di soddisfare una precisione elevata, esso convergerà ad un risultato inaccurato!

Vediamo questo comportamento in un esempio

Esercizio .23. [Newton, radici multiple] Si consideri il polinomio

$$p(x) = x^3 - 6x^2 + 9x - 4$$

.

1. Si calcolino le sue radici a mano.
2. Si usi il metodo di Newton (con opportuni guess iniziali differenti) per calcolare tutte le sue radici. Commentare.
3. Si usi il metodo di bisezione (in opportuni intervalli) per calcolare le sue radici. Commentare.

3.7.2 Metodo di Newton-Horner

Schema di Horner per la valutazione di un polinomio

Consideriamo il polinomio

$$p_{n-1}(x) = a_1 x^{n-1} + a_2 x^{n-2} + ... + a_{n-1}x + a_n.$$

Per la sua valutazione in un punto z occorrono $n-1$ addizioni e $\frac{n(n+1)}{2}$ moltiplicazioni. Se consideriamo invece la formulazione equivalente

$$p_{n-1}(x) = ((((a_1 x + a_2)x + a_3)x + ...)x + a_{n-1}) + a_n,$$

per la stessa valutazione bastano $n-1$ addizioni e $n-1$ moltiplicazioni. Dati il vettore *riga* a dei coefficienti e un vettore *colonna* z di valori, lo schema di Horner si implementa mediante l'algoritmo descritto dallo pseudocodice qui sotto, ove il vettore colonna `pnz=b(:,n)` contiene il valore del polinomio calcolato nei punti z. (Il comando di MATLAB polyval usa lo stesso schema di Horner.)

```
function [pnz,b] = horner(a,z)
%
% Scrivere delle specifiche dettagliate!
% [pnz,b] = horner(a,z)
%
n = length(a);
m = length(z);
b = zeros(m,n);
b(:,1) = a(1);
for j = 2:n
    b(:,j) = b(:,j-1).*z+a(j);
end
pnz = b(:,n);
```

Il polinomio (dipendente da z) definito da

$$q_{n-2}(x,z) = b_1 x^{n-2} + b_2 x^{n-3} + ... + b_{n-2}x + b_{n-1} \tag{3.8}$$

ove

$$\begin{aligned} b_1 &= a_1 \\ b_2 &= a_1 z + a_2 \\ &\dots \\ b_{n-1} &= ((a_1 z + a_2)z + \dots)z + a_{n-1} \\ b_n &= (((a_1 z + a_2)z + \dots)z + a_{n-1})z + a_n = p_{n-1}(z) \end{aligned}$$

è chiamato polinomio associato a $p_{n-1}(x)$. Valgono le seguenti uguaglianze:

$$\begin{aligned} p_{n-1}(x) &= b_n + (x-z)q_{n-2}(x,z) \\ p'_{n-1}(z) &= q_{n-2}(z,z) \end{aligned}$$

Se inoltre z è una radice di $p_{n-1}(x)$, allora $b_n = 0$ e $p_{n-1}(x) = (x-z)q_{n-2}(x,z)$ e dunque $q_{n-2}(x,z)$ ha per radici le restanti $n-2$ radici di $p_{n-1}(x)$.

Metodo di Newton–Horner per il calcolo delle radici di polinomi

Le considerazioni fatte al paragrafo precedente permettono di implementare in maniera efficiente il metodo di Newton per il calcolo di *tutte* le radici (anche complesse) dei polinomi. Infatti, dato il punto $x_1^{(k)}$ e $b_n^{(k)} = p_{n-1}(x_1^{(k)})$, l'algoritmo per calcolare $x_1^{(k+1)}$ (approssimazione $(k+1)$-esima della prima radice di $p_{n-1}(x)$) è

$$x_1^{(k+1)} = x_1^{(k)} - b_n^{(k)}/q_{n-2}^{(k)}(x_1^{(k)}, x_1^{(k)}) \tag{3.9}$$

ove $q_{n-2}^{(k)}(x_1^{(k)}, x_1^{(k)})$ è calcolato nuovamente mediante lo schema di Horner. Una volta arrivati a convergenza, diciamo con il valore $x_1^{K_1}$, il polinomio in x $q_{n-2}^{(K_1)}(x, x_1^{(K_1)})$ avrà le rimanenti radici di $p_{n-1}(x)$ e si applicherà nuovamente il metodo di Newton–Horner a tale polinomio. Tale tecnica è detta **deflazione** e si arresterà a $q_1^{(K_{n-2})}(x, x_{n-2}^{(K_{n-2})})$, per il quale $x_{n-1} = -b_2/b_1$. Per assicurare la convergenza alle radici complesse, è necessario scegliere una soluzione iniziale con parte immaginaria non nulla.

Esercizio .24 (Newton-Horner)**.** Si implementi il metodo di Newton–Horner e lo si testi per il calcolo di *tutte* le radici del polinomio $x^4 - x^3 + x^2 - x + 1$. Si confrontino i risultati con le radici ottenute dal comando di MATLAB roots. Commentare dettagliatamente il procedimento.

3.7.3 Metodo di Newton per radici multiple

Il metodo di Newton converge con ordine 1 alle radici multiple. Infatti, la funzione di iterazione g associata al metodo di Newton soddisfa

$$g'(\xi) = 1 - \frac{1}{m}$$

se ξ è una radice di molteplicità m. Tuttavia, spesso il valore di m non è noto a priori. Visto che comunque la successione $\{x(k)\}_k$ converge (linearmente) a ξ, si ha

$$\lim_{k\to\infty} \frac{x^{(k)} - x^{(k-1)}}{x^{(k-1)} - x^{(k-2)}} = \lim_{k\to\infty} \frac{g(x^{(k-1)}) - g(x^{(k-2)})}{x^{(k-1)} - x^{(k-2)}} = g'(\xi) = 1 - \frac{1}{m}$$

e dunque

$$\lim_{k\to\infty} \frac{1}{1 - \frac{x^{(k)} - x^{(k-1)}}{x^{(k-1)} - x^{(k-2)}}} = m.$$

Per recuperare la convergenza quadratica è necessario considerare il metodo

$$x^{(k+1)} = x^{(k)} - m\frac{f(x^{(k)})}{f'(x^{(k)})} \tag{3.10}$$

ove si sostituisce a m una sua stima. Tale stima può essere ottenuta applicando il metodo di Newton per radici semplici e calcolando le quantità

$$m^{(k)} = \frac{1}{1 - \frac{x^{(k)} - x^{(k-1)}}{x^{(k-1)} - x^{(k-2)}}} = \frac{x^{(k-1)} - x^{(k-2)}}{2x^{(k-1)} - x^{(k)} - x^{(k-2)}} \tag{3.11}$$

fino a quando $m^{(K)}$ è 'vicina' a $m^{(K-1)}$. A quel punto si applica il metodo di Newton per radici multiple (3.10) con $m = m^{(K)}$.

Esercizio .25 (Newton adattativo per radici multiple)**.** Si implementi il metodo di Newton adattivo per radici multiple (3.10). Lo si testi per il calcolo della radice multipla della funzione $f(x) = (x^2 - 1)^p \log x$, con $p = 2, 4, 6$, $x0 = 0.8$, tolleranza pari a 10^{-10} e numero massimo di iterazioni pari a 200. Lo si confronti con il metodo di Newton standard. Per la molteplicità m della radice provare sia considerandola nota a priori sia calcolandola come descritto in (3.11). Commentare lo svolgimento e i risultati ottenuti.

3.8 Metodo delle secanti

Il metodo delle secanti è una variazione del metodo di Newton. L'idea è quella di approssimare $f'(x_n)$ che appare nel metodo di Newton, con il rapporto incrementale

$$\frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}.$$

Si ottiene un metodo che si può scrivere nella seguente forma di punto fisso

$$x_{k+1} = x_k - f(x_k)\frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})} \tag{3.12}$$

con $f(x_{k-1}) \neq f(x_k)$. Il metodo richiede la conoscenza di due valori iniziali x_0, x_1.

Interpretazione geometrica. Al passo k, il nuovo valore x_{k+1} è l'intersezione della secante, ovvero la retta per i punti $(x_{k-1}, f(x_{k-1}))$ e $(x_k, f(x_k))$, con l'asse delle ascisse.

Confronto con Newton. Il metodo delle secanti converge con ordine di convergenza

$$p = \frac{1 + \sqrt{(5)}}{2}$$

inferiore quindi all'ordine di convergenza del metodo di Newton che è pari a 2.
Ma, l'importanza del metodo delle secanti sta principalmente nel costo computazionale: il metodo delle secanti richiede solo il calcolo di $f(x_k)$, mentre il metodo di Newton richiede sia il valore di $f(x_k)$, sia di $f'(x_k)$, e talvolta il calcolo della derivata di una funzione complessa può essere molto costoso. Comunque la scelta di un metodo piuttosto dell'altro dipende dalla particolare situazione.

Esercizio .26 (Metodo delle secanti)**.** Considerare la funzione

$$f(x) = x^2 - 2x - \log(x).$$

a) Plottare la funzione (in un intervallo coerente col suo dominio).

b) Calcolare entrambi gli zeri della funzione sia con il metodo delle secanti sia con il metodo di Newton; confrontare l'ordine di convergenza dei 2 metodi.

Suggerimento: per implementare la `function` per il metodo delle secanti modificare opportunamente quella usata per il metodo di Newton. (Per esempio fare in modo che accetti in input 2 valori iniziali, e che costruisca l'iterazione di punto fisso seguendo la formula in (3.12). Sistemare i commenti!)

Capitolo 4

Introduzione al calcolo matriciale

In questo capitolo ci occuperemo del calcolo di alcune grandezze direttamente collegate a vettori e matrici e cercheremo di capire che informazioni contengono. Nella parte finale inizieremo a costruire alcuni algoritmi base per risolvere i sistemi lineari. Vedremo anche nel corso dei prossimi capitoli che a seconda delle proprietà della matrice del sistema esistono metodi più o meno adatti.

4.1 Norme di vettori

Le principali norme di un vettore $x \in \mathbb{R}^n$ sono:

- $||x||_1 = \sum_{i=1}^{n} |x_i|$,
- $||x||_2 = \sqrt{\sum_{i=1}^{n} |x_i|^2}$,
- $||x||_\infty = \max_{1 \le i \le n} |x_i|$.

Esercizio .27. Per il calcolo delle norme di un vettore in $MATLAB$ si può usare il comando `norm`, con dei parametri opzionali $1, 2, \ldots,$ `inf`, per indicare quale norma si vuole calcolare (leggere anche l'`help`).

a) Creare una `function` per il calcolo delle norme di un vettore, senza usare il comando `norm`, e senza usare dei cicli.

   Suggerimento: usare i comandi `sum, sqrt, max`.

b) Confrontare i risultati ottenuti con la propria `function` con quelli ottenuti usando il comando $MATLAB$ `norm`, provandoli su qualche vettore a scelta.

4.2 Norme di matrici e numero di condizionamento

Le principali **norme di una matrice** $A \in \mathbb{R}^{n \times n}$ sono:

- $||A||_1 = \max_{1 \le k \le n} \sum_{i=1}^{n} |a_{ik}|$, (norma indotta da $||x||_1$),
- $||A||_\infty = \max_{1 \le i \le n} \sum_{k=1}^{n} |a_{ik}|$, (norma indotta da $||x||_\infty$),
- $||A||_2 = \sqrt{\rho(A^t A)}$, (norma indotta da $||x||_2$), dove ρ indica il raggio spettrale.

Il **numero di condizionamento** di una matrice si calcola con

$$\text{cond}(A, n) = ||A||_n ||A^{-1}||_n$$

dove n rappresenta una norma a piacere.

Esercizio .28. (Norme di matrici)

- Utilizzare il comando `norm` per calcolare le norme $1, 2$, e ∞ di due matrici a propria scelta. Usare il comando `cond` per calcolare anche il numero di condizionamento di tali matrici.
- Creare una `function` per il calcolo delle norme di una matrice, senza usare il comando `norm`, e usando al massimo un ciclo per ciascuna norma.

4.2.1 Matrice di Hilbert

La matrice $A^{(n)}$ di ordine n definita da

$$a_{ij}^{(n)} = \frac{1}{i+j-1} \quad i, j = 1, \ldots, n$$

è detta matrice di Hilbert. Per $n = 5$ si ha

$$A^{(5)} = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ & & \cdots & & \\ & & \cdots & & \\ \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} \end{bmatrix}.$$

Esercizio .29. Calcolare il condizionamento in norma infinito delle matrici di Hilbert per $n = 2, 3, 4, \ldots 10$. Fare un grafico `loglog` che mostri la crescita del numero di condizionamento: che tipo di crescita ha? (quadratica, esponenziale ... ?).

Suggerimenti: per generare automaticamente con $MATLAB$ una matrice di Hilbert si può utilizzare il comando `hilb`; Il grafico per il numero di condizionamento dovrebbe essere simile a quello riportato in Figura 4.1.

4.3 Effetto del numero di condizionamento sulla soluzione di sistemi lineari

Per vedere l'effetto del condizionamento di una matrice sulla soluzione di un sistema lineare $Ax = b$ si può considerare la seguente idea:

a) Sia data una matrice A,

b) Si fissi una soluzione di riferimento $x_{\mathbf{rif}}$ rispetto alla quale misurare gli errori relativi ed assoluti (in norma 2).

[Suggerimento: Per fissare una soluzione di riferimento data la matrice A, si sceglie a piacere un vettore $x_{\mathbf{rif}}$ e si determina poi il termine noto b come prodotto tra A ed $x_{\mathbf{rif}}$].

c) Si risolva il sistema lineare $Ax = b$ (usando il comando $MATLAB$ `x=A\b`),

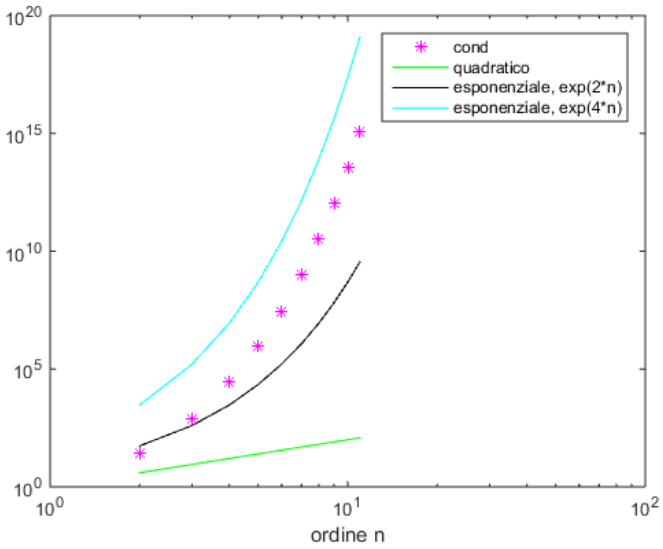

Figura 4.1: Crescita del numero di condizionamento per la matrice di Hilbert di ordine n.

d) Calcolare gli errori relativi ed assoluti tra x e $x_{\mathbf{rif}}$.

e) Perturbare il termine b con un piccolo errore e quindi costruire un nuovo vettore $\hat{b}$,

f) Risolvere il nuovo sistema lineare $A\hat{x} = \hat{b}$,

g) Calcolare gli errori (relativo ed assoluto) tra $\hat{x}$ e $x_{\mathbf{rif}}$.

Esercizio .30. Testare l'idea presentata qui sopra (fasi a-g) prima su una matrice ben-condizionata (a scelta, esempio: matrice tridiagonale simmetrica con elementi sulla diagonale pari a 1 ed elementi sulla sovra e sotto diagonale generati in modo casuale) e poi su una matrice mal condizionata (come quella di Hilbert). Commentare i risultati ottenuti.

4.4 Sistemi a risoluzione immediata

4.4.1 Sistema diagonale

Il sistema lineare $Ax = b$ con A matrice diagonale della forma

$$\begin{bmatrix} a_{11} & & & \\ & a_{22} & & \\ & & \dots & \\ & & & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix},$$

he per soluzione il vettore $x = (x_1, x_2, \dots, x_n)$ con

$$x_i = \frac{b_i}{a_{ii}}, \quad i = 1, \dots, n.$$

Esercizio .31. Scrivere una `function` in $MATLAB$ per risolvere un sistema diagonale. Scrivere delle specifiche complete e ordinate. Si scelga poi a piacere una matrice diagonale e un vettore dei termini noti, e si testi il corretto funzionamento del proprio algoritmo.

Suggerimenti:

- L'algoritmo risolutivo deve ricevere in ingresso la matrice diagonale A, il vettore dei termini noti b e deve restituire in uscita il vettore della soluzione x.
- L'algoritmo può essere implementato o usando un ciclo, oppure usando la divisione puntuale tra il vettore b e la diagonale di A (scegliere il metodo che si preferisce).
- Per costruire un esempio di sistema su cui testare l'algoritmo per il quale si sappia già la soluzione esatta si può seguire il suggerimento in **b** nella sezione 4.3.

4.4.2 Sistema triangolare inferiore

Il sistema lineare $Ax = b$ con A matrice triangolare inferiore della forma

$$\begin{bmatrix} a_{11} & & & \\ a_{21} & a_{22} & & \\ a_{31} & a_{32} & a_{33} & \\ \dots & \dots & \dots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix},$$

he per soluzione il vettore $x = (x_1, x_2, \dots, x_n)$ ottenuto dall'algoritmo detto delle *sostituzioni in avanti* corrispondente a

$$\begin{aligned} x_1 &= \frac{b_1}{a_{11}} \\ x_i &= \frac{1}{a_{ii}}(b_i - \sum_{k=1}^{i-1} a_{ik}x_k) \quad i = 2, \dots, n. \end{aligned} \tag{4.1}$$

Esercizio .32. Scrivere una `function` in $MATLAB$ per risolvere un sistema triangolare inferiore. Scrivere delle specifiche complete e ordinate. Si scelga poi a piacere una matrice triangolare inferiore e un vettore dei termini noti, e si testi il corretto funzionamento del proprio algoritmo.

Suggerimento: Seguire la formula in (4.1) e implementarla attraverso due cicli `for` annidati.

4.4.3 Sistema triangolare superiore

Il sistema lineare $Ax = b$ con A matrice triangolare superiore della forma

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ & a_{22} & \dots & a_{2n} \\ & & \dots & \\ & & & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix},$$

he per soluzione il vettore $x = (x_1, x_2, \ldots, x_n)$ ottenuto dall'algoritmo detto delle *sostituzioni all'indietro* corrispondente a

$$\begin{aligned} x_n &= \frac{b_n}{a_{nn}} \\ x_i &= \frac{1}{a_{ii}}(b_i - \sum_{k=i+1}^{n} a_{ik}x_k) \quad i = n-1, \ldots, 1. \end{aligned} \tag{4.2}$$

Esercizio .33. Scrivere una `function` in MATLAB per risolvere un sistema triangolare superiore. Scrivere delle specifiche complete e ordinate. Si scelga poi a piacere una matrice triangolare superiore e un vettore dei termini noti, e si testi il corretto funzionamento del proprio algoritmo.

Suggerimento: Seguire la formula in (4.2) e implementarla attraverso due cicli `for` annidati: quello esterno deve essere un ciclo discendente.

4.4.4 Sistema con matrice ortogonale

In tali casi la matrice A è ortogonale, ovvero $A^{-1} = A^T$, pertanto la soluzione di $Ax = b$ è data da $x = A^T b$.

Esercizio .34. Scrivere una `function` in $MATLAB$ per risolvere un sistema con matrice ortogonale. Scrivere delle specifiche complete e ordinate.

Si scelga poi a piacere una matrice ortogonale e un vettore dei termini noti, e si testi il corretto funzionamento del proprio algoritmo. (Prima di eseguire l'algoritmo verificare che la matrice scelta sia davvero ortogonale).

Richiami di Matlab. In $MATLAB$ ci sono delle `routine` predefinite per creare matrici specifiche. In particolare si può guardare la documentazione completa del comando `gallery` dove, tra le altre matrici descritte, se ne trovano anche 6 ortogonali (guardare nella sezione `orthog`). Scrivendo `gallery('orthog', n, k)` si può scegliere la dimensione della matrice e il tipo.

Proprietà. Si può dimostrare che la norma 2 di una matrice ortogonale è sempre 1. Di conseguenza anche il suo numero di condizionamento vale sempre 1. Quindi, per esempio, quando moltiplichiamo la matrice di un sistema lineare (e il termine noto) per una matrice ortogonale non peggioriamo il condizionamento del sistema.

Esercizio .35. Costruire qualche matrice ortogonale (scegliendole dalla `gallery` di $MATLAB$ come suggerito sopra) e verificare che esse hanno norma 2 e numero di condizionamento, in norma 2, pari a 1.

Verificare inoltre che moltiplicando una matrice qualsiasi per una matrice ortogonale la sua norma 2 e il suo numero di condizionamento rimangono invariati. Commentare.

Capitolo 5

Metodi diretti per la soluzione di un sistema lineare

Per risolvere un sistema lineare esistono diversi metodi tra cui scegliere a seconda della situazione e del tipo di matrice. Nel capitolo precedente abbiamo trattato alcune matrici con una struttura molto particolare: matrici diagonali, triangolari o ortogonali. Per queste esistono algoritmi di soluzione immediati, che ci evitano sia di calcolare l'inversa, sia di applicare procedure costose.

Quando invece si considera una matrice generica, esistono sostanzialmente due classi di metodi:

- I *metodi diretti* sono quelli in cui si calcola una fattorizzazione della matrice iniziale in matrici più semplici (come appunto le matrici diagonali, ortogonali o triangolari) per le quali risolvere il sistema associato sia estremamente rapido.
- I *metodi indiretti* nei quali si parte da una soluzione approssimata x_0 e tramite successive iterazioni, (che in generale mirano a minimizzare la norma del residuo), si cerca di determinare una soluzione sempre più vicina a quella esatta.

Si opta per un metodo indiretto quando la matrice del sistema è di grandi dimensioni ed in particolare se è *sparsa*.

Si dice *sparsa* una matrice di dimensione $n \times n$ in cui la quantità di elementi non nulli è dell'ordine di $O(n)$. In tal caso per esempio memorizzare la matrice ha un costo ridotto perché vengono memorizzati solo gli elementi non nulli. Però in generale data una matrice sparsa, le matrici ottenute fattorizzandola non sono sparse e il costo di memorizzazione aumenta: è questo uno dei motivi per cui non si sceglie di usare un metodo diretto per risolvere sistemi con matrici sparse.

Una situazione in cui invece si preferiscono i metodi diretti è quella in cui si debbano risolvere più sistemi in cui la matrice è fissata e cambia solo il termine noto. In questo caso infatti l'operazione più costosa, cioè la scomposizione, deve essere fatta una sola volta, e poi rimarranno da risolvere solo sistemi immediati.

N.B. In generale non è consigliabile risolvere un sistema mediante il calcolo diretto dell'inversa della matrice del sistema. Infatti gli algoritmi per il calcolo dell'inversa (e quindi il comando `inv` di Matlab) sono fortemente malcondizionati.

Vediamo ora le principali scomposizioni che possiamo applicare alle matrici e confrontiamo le loro proprietà e il loro costo computazionale.

5.1 Decomposizione ai valori singolari, SVD

Data una matrice $A \in \mathbb{R}^{n\times n}$, esistono due matrici $P, Q \in R^n$ unitarie (cioè ortogonali e con modulo del determinante uguale e 1), e una matrice $D \in R^n$ diagonale, tale che $A = PDQ^t$. In particolare sulla diagonale della matrice D si trovano i valori $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n$ detti valori singolari di A.

I *valori singolari* di una matrice A sono le radici degli autovalori della matrice A^tA.

La decomposizione ai valori singolari, chiamata in inglese *Singular Value Decomposition* si ottiene in $MATLAB$ con il comando $[P, D, Q] =$ `svd(A)`.

Esercizio .36 (Proprietà della SVD)**.** Si calcoli, usando l'apposito comando $MATLAB$, la decomposizione ai valori singolari della matrice di Hilbert di ordine 5 e di una matrice triangolare superiore di dimensione 4×4 a scelta. Si verifichi (numericamente) per entrambi i casi che sono rispettate le seguenti proprietà:

a) Le matrici P e Q sono unitarie;

b) La matrice A si ottiene davvero come PDQ^t;

c) D è una matrice diagonale e sulla sua diagonale ci sono i *valori singolari* di A. (Suggerimento: si ricordi che per calcolare gli autovalori di una matrice esiste il comando `eig`).

d) $det(A) = \pm \prod_{i=1}^{n} \sigma_i$;

e) $||A||_2 = \sigma_1$;

f) $cond_2(A) = \frac{\sigma_1}{\sigma_n}$.

5.1.1 Applicazione della SVD alla soluzione di un sistema lineare

La decomposizione di una matrice in sottomatrici che abbiano particolari proprietà è utile per la risoluzione di sistemi lineari. Infatti al posto di risolvere il sistema lineare $Ax = b$, se ho a disposizione la scomposizione di A in PDQ^t, posso risolvere

$$PDQ^t x = b. \tag{5.1}$$

La soluzione di tale sistema può essere spezzata in più fasi

- Definisco il vettore
$$z := DQ^t x, \tag{5.2}$$
riscrivo (8.2) come $Pz = b$. P è una matrice ortogonale quindi ricavo z semplicemente facendo $z = P^t b$.

- Definisco il vettore
$$y := Q^t x, \tag{5.3}$$
così posso riscrivere la (5.2) come $Dy = z$ e siccome D è una matrice diagonale ottengo y semplicemente come $y = z./diag(D)$.

- Infine posso trovare x risolvendo la (5.3), cioè risolvendo $Q^t x = y$, e siccome di nuovo Q è ortogonale, x si ottiene facendo $x = Qy$.

La risoluzione di sistemi con matrici ortogonali oppure diagonali è molto efficiente, ma bisogna ricordare che al contrario il calcolo della decomposizione della matrice può essere molto costoso. Quindi tale metodo risolutivo può essere impiegato quando la decomposizione della matrice è già nota, oppure quando si devono risolvere più sistemi che differiscono tra loro solo per il termine noto. In quest'ultimo caso infatti basta decomporre la matrice una sola volta.

Esercizio .37 (SVD per risolvere un sistema)**.** Si consideri una matrice A random abbastanza grande (per esempio 100×100) non singolare. Si costruisca una soluzione di riferimento $xRif$ e un termine noto b a piacere.

Si supponga nota (cioè si calcoli all'inizio dell'algoritmo) la decomposizione SVD della matrice e, la trasposta della matrice P.

a) Si risolva il sistema $Ax = b$ (usando il comando apposito di $MATLAB$: \ ): si calcoli (con i comandi `tic, toc`) il tempo impiegato e la norma infinito dell'errore assoluto commesso.

b) Poi si risolva il sistema equivalente riportato in (8.2) seguendo le varie fasi descritte nell'elenco puntato qui sopra: si calcoli il tempo impiegato dalle tre fasi (insieme) e la norma infinito dell'errore assoluto commesso.

c) Separatamente si calcoli il tempo impiegato da $MATLAB$ per eseguire la SVD.

d) Si commentino e confrontino i risultati ottenuti. E' utile decomporre la matrice? In che situazioni? Perché?

Nota. Negli esercizi fatti precedentemente avevamo costruito delle routine apposta per risolvere sistemi diagonali e ortogonali. Vista la loro semplicità, invocarle è troppo costoso per poter anche vedere i miglioramenti apportati da questa procedura: quindi in questo caso preferiamo riscrivere il codice che risolve i sistemi direttamente dentro l'esercizio.

5.2 Fattorizzazione QR di una matrice

Data una matrice $A \in R^{m\times n}$, $(m \geq n)$ esiste una matrice ortogonale $Q \in \mathbb{R}^{m\times m}$ tale che

$$A = QR = Q \begin{bmatrix} R_1 \\ 0 \end{bmatrix},$$

dove $R \in \mathbb{R}^{m\times n}$ con $r_{ij} = 0$ se $i > j$ cioè $R_1 \in R^{n\times n}$ è triangolare superiore.

Poiché la matrice Q è ortogonale e, se A è quadrata, $R = R_1$ è triangolare superiore, i sistemi associati a queste matrici sono facili da risolvere. Quindi anche in questo caso, nota la fattorizzazione QR di una matrice, si preferisce risolvere un sistema del tipo $Ax = b$, anziché in modo diretto, con i passaggi seguenti:

$$A = QR \quad \Rightarrow \quad Ax = QRx = b \quad \Rightarrow \quad \begin{cases} Qy = b & \Rightarrow \quad y = Q^t b \\ Rx = y \end{cases} \tag{5.4}$$

Esercizio .38 (QR per risolvere un sistema)**.** Risolvere il sistema $Ax = b$ con

$$A = \begin{bmatrix} 4 & 1 & 0 & 1 \\ 1 & 4 & 1 & 0 \\ 0 & 1 & 4 & 1 \\ 1 & 2 & 1 & 4 \end{bmatrix}$$

mediante l'uso della fattorizzazione QR, cioè seguendo la procedura in (5.4) che porta a risolvere un sistema ortogonale e uno triangolare superiore. Scegliere b in modo che la soluzione di riferimento sia un vettore di tutti 1. Commentare le fasi della soluzione.

Suggerimento: per calcolare la fattorizzazione QR della matrice si può usare il comando `qr` di $MATLAB$; per risolvere il sistema triangolare superiore si utilizzi la routine implementata nella scorsa lezione.

5.2.1 Fattorizzazione QR e minimi quadrati

Siano dati m punti (xP_i, yP_i), $0 \leq i \leq m$. Sia $p(x) = ax + b$ il polinomio lineare di grado $n = 1$ con cui vogliamo approssimare i dati.

Il problema dei minimi quadrati consiste nel rendere minima la seguente quantità (scarti al quadrato)

$$S = \sum_{i=0}^{m} (p(xP_i) - yP_i)^2 = \sum_{i=0}^{m} (axP_i + b - yP_i)^2.$$

Riscriviamo il problema in forma matriciale: *Trovare* $z = [a, b]$ *affinché la quantità seguente sia minima*

$$||Az - yP||_2 = \left|\left| \begin{bmatrix} xP_0 & 1 \\ xP_1 & 1 \\ xP_2 & 1 \\ \vdots & \vdots \\ xP_n & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} - \begin{bmatrix} yP_0 \\ yP_1 \\ yP_2 \\ \vdots \\ yP_n \end{bmatrix} \right|\right|_2 . \tag{5.5}$$

Tale problema di minimo può essere risolto, per esempio, utilizzando le *equazioni normali* oppure la *fattorizzazione QR*.

Esercizio .39. Siano date le seguenti coppie di valori

xP_i	-5	-3	1	3	4	6	8
yP_i	18	7	0	7	16	50	67.

Si vuole calcolare la retta di regressione $p = ax + b$ usando sia la soluzione delle equazioni normali, sia la fattorizzazione QR e poi disegnarla. Procedere nel modo seguente:

Equazioni Normali

a) Scrivere una routine dal nome `CostruisciA` che costruisca la matrice A della retta di regressione ai minimi quadrati, come definita nell'equazione (5.5). `CostruisciA` riceve in ingresso il vettore xP e restituisce in uscita la matrice A.

b) Scrivere una routine `SistNormale` che costruisca la matrice A^tA e il vettore $\hat{b} = A^tyP$ delle equazioni normali ($A^tAz = A^tyP$). La soluzione del sistema sarà il vettore $z = [a, b]$ contenente il coefficiente angolare a e il termine noto b della retta di regressione.

```
1    function [AtA,b] = SistNormale(xP,yP)
2        A <- CostruisciA(xP)
3        AtA <- A^t*A
4        b <- A^t*yP
```

c) Risolvere il sistema $A^tAz = A^tyP$ mediante la routine \ di $MATLAB$.

Fattorizzazione QR Per usare la fattorizzazione QR procedere come nel seguente pseudocodice

```
\\ calcola fattorizzazione QR della matrice A (precedentemente
   costruita mediante la routine CostruisciA)
[Q,R] <- qr (A)
c <- Q^t * yP
\\ (scegli n in modo che R1 possa essere quadrata)
c1 <- (c1, .. , cn)^t
R1 <- R(1, .. ,n; 1,.., n) \\ le prime n righe e colonne di R

\\ calcolare soluzione di R1 z = c1 (che e' un sistema triangolare superiore)
   mediante sostituzione all'indietro (usare l'apposito codice
   implementato nella scorsa lezione)
z <-  ......
```

Visualizzazione dei risultati Visualizzare i risultati ottenuti, tracciando nello stesso grafico i punti (xP_i, yP_i), e le rette $p = ax + b$ i cui coefficienti a, b sono racchiusi nei vettori z calcolati con le due procedure (equazioni normali e QR) svolte nei passi precedenti.

I risultati ottenuti dovrebbero essere simili a quelli riportati in Figura 5.1. Fare specifiche e commenti ordinati e completi.

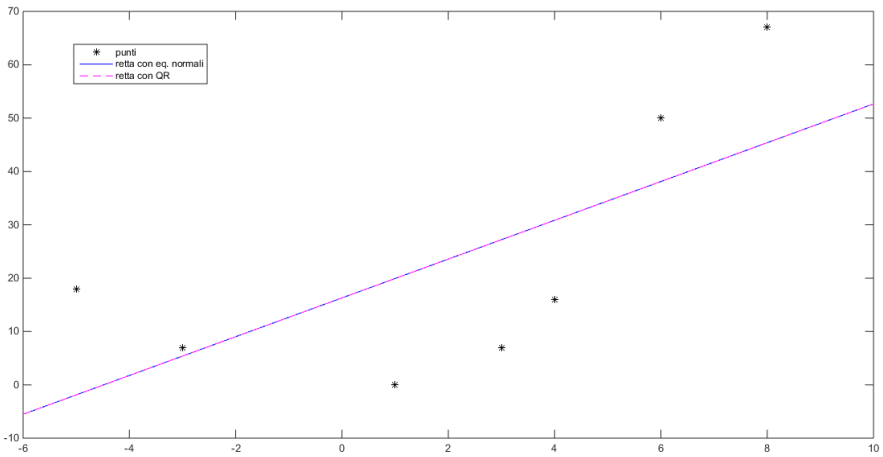

Figura 5.1: Esercizio .39 : retta di regressione.

5.3 Decomposizione LU con pivoting

Data una matrice $A \in \mathbb{R}^{n\times n}$ invertibile esistono

- una matrice P, detta di *permutazione*,
- una matrice L *triangolare inferiore*, con elementi sulla diagonale tutti uguali ad 1

- una matrice U *triangolare superiore*

tali che

$$PA = LU\,.$$

In $MATLAB$ tale decomposizione si ottiene con il comando `[L,U,P] = lu(A)`.
Attenzione a non dimenticare la matrice P quando si usa il comando MATLAB: infatti MATLAB esegue sempre la decomposizione con pivoting, anche se ci si dimentica di salvare tra gli output la matrice P.

5.3.1 Osservazioni

- Dato un sistema lineare $Ax = b$, e la decomposizione LU con pivoting della matrice A, al posto di risolvere direttamente il sistema $Ax = b$ si possono risolvere dei sistemi triangolari secondo la seguente strategia

$$Ax = b \quad \Rightarrow \quad LUx = Pb \quad \Rightarrow \quad \begin{cases} Ly = Pb & \Rightarrow \quad \text{sistema triang. inf} \\ Ux = y & \Rightarrow \quad \text{sistema triang. sup} \end{cases} \tag{5.6}$$

- Grazie a tale fattorizzazione è facile calcolare il modulo del determinante della matrice A, infatti, usando il teorema di Binet, il fatto che il determinante di una matrice triangolare è dato dal prodotto dei valori sulla diagonale, e infine che sulla diagonale di L ci sono tutti 1 si ottiene

$$|det(A)| = |det(LU)| = |(det(L)(det(U)| = |\texttt{prod(diag(U))}|. \tag{5.7}$$

  Nota. Il segno del determinante dipende da se il numero di permutazioni eseguite è pari o dispari.

Esercizio .40. a) Data la matrice

$$A = \begin{bmatrix} 4 & 1 & 0 & 1 \\ 1 & 4 & 1 & 0 \\ 0 & 1 & 4 & 1 \\ 1 & 2 & 1 & 4 \end{bmatrix},$$

calcolarne la fattorizzazione LU con pivoting, usando l'apposito comando di $MATLAB$.

b) Verificare che il calcolo del determinante di A può essere eseguito come descritto in (5.7).

c) Considerare il sistema $Ax = b$, scegliere b in modo che la soluzione di riferimento sia un vettore di tutti 1; risolvere tale sistema sfruttando la strategia descritta in (5.6).

d) Scegliere qualche matrice random (invertibile) e confrontare, tramite i comandi `tic, toc`, il tempo necessario per ottenere le tre diverse fattorizzazioni studiate (SVD, QR ed LU) tutte calcolate usando gli appositi comandi $MATLAB$ (`svd, qr, lu`). Commentare.

5.4 Fattorizzazione di Cholesky

Data una matrice A quadrata che sia

- simmatrica: $A^t = A$
- definita positiva: $\forall x \in \mathbb{R}^n \; x^t A x \geq 0$

è possibile calcolare una matrice L triangolare inferiore tale che

$$A = L^t L.$$

Come nei casi precedenti una volta scomposta la matrice A, al posto di risolvere il sistema $Ax = b$ si possono risolvere due sistemi più semplici: uno con matrice triangolare inferiore ed uno con matrice triangolare superiore.

5.4.1 Lati positivi e negativi, in sintesi

L'algoritmo di fattorizzazione di Cholesky è due volte più veloce di quello standard per la fattorizzazione LU e non richiede mai l'introduzione del pivoting. Inoltre dovendo risolvere un sistema lineare, la soluzione ottenuta risolvendo i sottoproblemi associati alla fattorizzazione di Cholesky risulta più accurata e stabile rispetto a quella ottenuta con la *LU* (per tutti i dettagli si veda Comincioli *Analisi numerica: metodi, modelli, applicazioni*).

Però questa tecnica può essere usata solo quando la matrice è simmetrica e definita positiva, e nel caso in cui questa informazione non sia già nota (per esempio perché è stata costruita apposta), il modo più veloce per verificare che lo sia è proprio applicare la fattorizzazione stessa e vedere se va a buon fine. Inoltre l'algoritmo è si più veloce di quello per la LU ma solo di una costante moltiplicativa.

É quindi compito nostro scegliere quale sia il metodo più efficiente a seconda della situazione.

Esercizio .41 (Fattorizzazione di Cholesky)**.** Si consideri la matrice di Hilbert di ordine 4.

a. Si confrontino i tempi necessari per calcolare le fattorizzazioni LU e Cholesky di questa matrice.

b. Si risolva il sistema lineare associato a tale matrice (scegliendo il termine noto b in modo che la soluzione di riferimento sia un vettore di tutti 1) con entrambi i metodi, e si confronti la norma infinito dell'errore.

c. Ripetere i punti precedenti con la matrice di Hilbert di ordine 14. Commentare.

Capitolo 6

Metodi indiretti per la soluzione di un sistema lineare

I metodi indiretti (o metodi iterativi) sono basati sul calcolo della soluzione $\mathbf{x}$ del sistema lineare $A\mathbf{x} = b$ come limite di una successione convergente $\{\mathbf{x}^{(k)}\}$, cioè

$$\lim_{k\to\infty} \mathbf{x}^{(k)} = \mathbf{x}.$$

Una classe particolare di iterazioni sono della forma

$$\mathbf{x}^{(k+1)} = B\mathbf{x}^{(k)} + q$$

dove $B \in \mathbb{R}^{n\times n}$ è detta matrice di iterazione e $q \in R^n$ è un vettore.

Il seguente teorema ne caratterizza la convergenza

Teorema 6.1 (Convergenza - Condizione necessaria e sufficiente)**.** Condizione necessaria e sufficiente affinché la successione $\mathbf{x}^{(k)}$ converga a $\mathbf{x}$ è che il raggio spettrale di B, $\rho(B)$, sia minore di 1.

Nota. La notazione $\rho(B)$ indica il raggio spettrale della matrice B, cioè il modulo massimo degli autovalori di B.

6.1 Metodo di Jacobi

Per prima cosa, scriviamo il metodo di Jacobi in *forma matriciale*. Consideriamo la decomposizione di una matrice A in

$$A = D + L + U \tag{6.1}$$

con

- D, la diagonale di A
- L, la parte triangolare inferiore di A (esclusa la diagonale)
- U, la parte triangolare superiore di A (esclusa la diagonale).

La *matrice di iterazione* B_J corrispondente a questo metodo è

$$B_J = -D^{-1}(L+U) \tag{6.2}$$

e il corrispondente schema iterativo è

$$\mathbf{x}^{(k+1)} = -D^{-1}(L+U)\mathbf{x}^{(k)} + D^{-1}b.$$

Scrivendo invece l'iterazione *per componenti* si ottiene

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1, j\neq i}^{n} a_{ij}x_j^{(k)}\right), \quad i = 1, \ldots, n. \tag{6.3}$$

Teorema 6.2 (Convergenza - Condizione sufficiente)**.** Condizione sufficiente affinché il metodo di Jacobi converga a $\mathbf{x}$ è che la matrice A del sistema sia dominante per righe in senso stretto.

6.2 Metodo di Gauss-Seidel

Data la decomposizione della matrice A come in (6.1) la *matrice di iterazione* del metodo di Gauss-Seidel è

$$B_{GS} = -(D+L)^{-1}U \tag{6.4}$$

il corrispondente schema iterativo è

$$\mathbf{x}^{(k+1)} = -(D+L)^{-1}U\mathbf{x}^{(k)} + (D+L)^{-1}b.$$

Scrivendo invece l'iterazione *per componenti* si ottiene

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}\right), \quad i = 1, \ldots, n. \tag{6.5}$$

Teorema 6.3 (Convergenza - Condizione sufficiente)**.** Condizione sufficiente affinché il metodo di Gauss Seidel converga è che la matrice A del sistema sia dominante per righe in senso stretto.

Teorema 6.4 (Convergenza - Condizione sufficiente)**.** Condizione sufficiente affinché il metodo di Gauss Seidel converga a $\mathbf{x}$ è che la matrice A del sistema sia definita positiva.

6.3 Test di arresto

I metodi presentati sono iterativi: ciò significa che a partire da una soluzione approssimata (un qualsiasi *guess* iniziale) ne cercano, con successive iterazione, una sempre più vicina a quella reale (se convergono). Bisogna quindi stabilire un criterio detto *criterio d'arresto* che quando risulta verificato fa interrompere questa ricerca. Vi sono due test principali che possono essere utilizzati:

- Test sullo *scarto* tra due iterate: si potrà fermare un metodo iterativo all'iterazione $k+1$ quando $||x^{(k+1)} - x^{(k)}||_\infty < tol$
- Test sul *residuo*: si potrà fermare un metodo all'iterazione k quando $\frac{||r^{(k)}||_2}{||b||_2} < tol$ dove $r^{(k)} = b - Ax^{(k)}$.

In entrambi i casi il test può essere reso più accurato introducendo un criterio d'arresto misto, basato sia su una tolleranza assoluta sia su una tolleranza relativa, vedi Sezione 3.3.1.

6.4 Esercizi

Esercizio .42 (Convergenza metodi iterativi)**.** Notare che le seguenti matrici non sono dominanti per righe e che quindi le due facili condizioni sufficienti per la convergenza di Jacobi e Gauss Seidel non possono essere applicate.
Disegnare quindi, per ognuna delle seguenti matrici

$$A1 = \begin{pmatrix} 3 & 0 & 4 \\ 7 & 4 & 2 \\ -1 & 1 & 2 \end{pmatrix} \quad A2 = \begin{pmatrix} -3 & 3 & -6 \\ -4 & 7 & -8 \\ 5 & 7 & -9 \end{pmatrix} \quad A3 = \begin{pmatrix} 4 & 1 & 1 \\ 2 & -9 & 0 \\ 0 & -8 & 6 \end{pmatrix} \quad A4 = \begin{pmatrix} 7 & 6 & 9 \\ 4 & 5 & -4 \\ -7 & -3 & 8 \end{pmatrix}$$

- lo spettro di A
- lo spettro della matrice di iterazione di Jacobi B_J, definita in (6.2)
- lo spettro della matrice di iterazione di Gauss-Seidel B_{GS}, definita in (6.4)

Dire inoltre, analizzando gli spettri di tali matrici, se tali metodi potranno convergere per qualunque valore di $\mathbf{x}_0$ (facendo riferimento al Teorema 6.4).

Suggerimenti: Per estrarre la diagonale, la parte triangolare inferiore o superiore di una matrice considerare l'uso dei comandi `diag, tril, triu`.

Per spettro si intende l'insieme degli autovalori di una matrice; essi devono essere rappresentati sul piano complesso (con i comandi `real` e `imag` si estraggono le parti reali e immaginarie di un numero complesso). Ad esempio per la matrice $A1$ si dovrebbe ottenere un risultato simile a quello riportato in Figura 6.1.

Esercizio .43 (Jacobi)**.** Scrivere una `routine` $MATLAB$ che implementi l'iterazione di Jacobi in (6.3). Provare tale routine sulle matrici dell'esercizio precedente scegliendo il termine noto b in modo che la soluzione di riferimento sia fatta da tutti 1.

Suggerimenti per l'implementazione della `routine` per Jacobi:

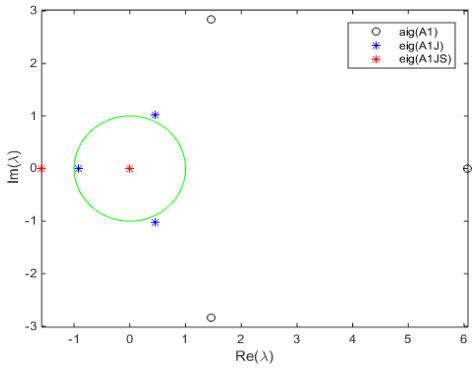

Figura 6.1: Rappresentazione dello spettro d A1.

- La function deve prendere come argomenti in **input** la matrice del sistema A, il termine noto b, un'approssimazione iniziale della soluzione x_0, la tolleranza con cui si vuole cercare la soluzione `tol`, e il numero massimo di iterazioni che concediamo al metodo `maxiter`.
- La function deve restituire in **output**: la soluzione del sistema (a meno della tolleranza concessa) x, una variabile booleana che indichi se il metodo è arrivato a convergenza oppure no, e una matrice `xhist` sulle cui colonne siano presenti tutte le soluzioni calcolate durante le varie iterazioni (in particolare sulla prima colonna deve esserci x_0 e sull'ultima x).
- Prima fase: inizializzare `xhist`, la variabile booleana, il contatore delle iterazioni e una variabile `scarto` per il test di arresto (vedi sezione 6.3).
- Seconda fase: costruzione di un ciclo `while` che terminerà o una volta trovata una soluzione abbastanza precisa (che supera il test dello scarto), o al raggiungimento del numero massimo di iterazioni concesse.
- All'interno del ciclo `while` bisogna implementare, attraverso dei cicli `for` la formula in (6.3): la quale permette di trovare ciascuna componente della nuova soluzione a partire dalle componenti della vecchia.
- Aggiornare le variabili `scarto`, x, e `xhist`.
- Dopo il ciclo `while` occuparsi della variabile booleana che indica la convergenza o meno del metodo e se necessario di `xhist`
- Testare per prima cosa il metodo su una matrice, su cui si è certi che il metodo convergerà.

Esercizio .44 (Gauss-Seidel)**.** Scrivere una `routine` $MATLAB$ che implementi l'iterazione di Gauss-Seidel in (6.5). Provare tale routine sulle matrici dell'esercizio 2 scegliendo il termine noto b in modo che la soluzione di riferimento sia fatta da tutti 1.

Suggerimenti per l'implementazione della `routine` per Gauss-Seidel: la `routine` è sostanzialmente uguale a quella di Jacobi, l'unica differenza sta nel fatto che per calcolare la componente $x(i)$ della nuova soluzione invece di usare tutte le vecchie componenti si usano anche quelle nuove, se si sono già trovate: cioè si usano le $x(1 : i-1)$ nuove e le $x(i+1 : n)$ vecchie.

Esercizio .45 (Confronto Jacobi-GaussSeidel, matrice differenze finite, autonomamente)**.** La seguente matrice è usata nel contesto del metodo delle *differenze finite* per risolvere un'equazione differenziale, in particolare si usa per discretizzare le derivate seconde

```
B = (1/h^2)*toeplitz(sparse(1,[1,2],[-2,1],1,n),sparse(1,[1,2],[-2,1],1,n));
```

Scegliere per esempio $n = 10$ e $h = 0.1$.

a) Studiare la convergenza del metodo di Jacobi e di Gauss-Seidel su questa matrice analizzando lo spettro delle matrici d'iterazione (il comando `eigs` calcola i 6 autovalori in modulo maggiori di una matrice sparsa).

b) Scegliere un termine noto b in modo che la soluzione di riferimento del sistema sia fatta da tutti 1. Risolvere il sistema sia con Jacobi che con Gauss-Seidel.

c) Servendosi della matrice `xhist` restituita dai due metodi, tracciare un grafico in scala semi-logaritmica dell'andamento dell'errore.

I risultati ottenuti dovrebbero essere simili a quelli riportati in Figura 6.2. (Il numero di iterazioni dipende molto dalla tolleranza e dall'approssimazione iniziale scelte. Inoltre usando come soluzione di riferimento quella costruita apposta nel secondo grafico della Figura vengono proprio 2 rette; se invece si usa come soluzione di riferimento l'ultima trovata dal vostro metodo allora si ottiene il grafico curvo come mostrato nella Figura.).

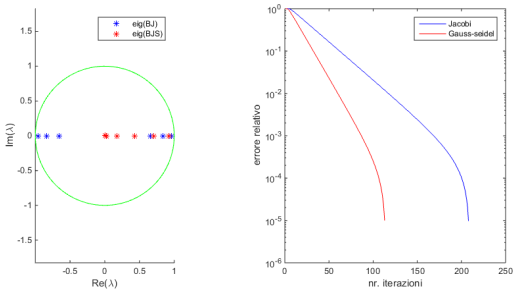

Figura 6.2: Spettro della matrice delle derivate seconde e analisi della convergenza.

6.5 Il Metodo di SOR

Il metodo SOR (Successive Over-Relaxation) è una generalizzazione dell'iterazione di Gauss-Seidel. Infatti ricordando l'iterazione di Gauss-Seidel

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}\right), \quad i = 1, \ldots, n,$$

si aggiunge e sottrae la componente $x_i^{(k)}$ ottenendo

$$x_i^{(k+1)} = x_i^{(k)} + \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i}^{n} a_{ij}x_j^{(k)}\right), \quad i = 1, \ldots, n.$$

Ora si introduce il parametro reale ω di accelerazione nel seguente modo:

$$x_i^{(k+1)} = x_i^{(k)} + \frac{\omega}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i}^{n} a_{ij}x_j^{(k)}\right), \quad i = 1, \ldots, n.$$

e si può riscrivere nella forma

$$x_i^{(k+1)} = (1-\omega)x_i^{(k)} + \frac{\omega}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}\right), \quad i = 1, \ldots, n. \quad (6.6)$$

Per $\omega < 1$ si ha il metodo di sottorilassamento, per $\omega = 1$ il metodo coincide con l'iterazione di Gauss-Seidel mentre per $\omega > 1$ abbiamo il metodo di sovrarilassamento.

La formula precedente scritta in forma matriciale diventa

$$\mathbf{x}^{(k+1)} = (\omega L + D)^{-1}[(1-\omega)D - \omega U]\mathbf{x}^{(k)} + \omega(\omega L + D)^{-1}b.$$

in cui la matrice di iterazione risulta (dipendente dal parametro ω)

$$B_\omega = (\omega L + D)^{-1}[(1-\omega)D - \omega U] \quad (6.7)$$

Esercizio .46 (SOR)**.**

a) Scrivere una `routine` $MATLAB$ che implementi l'iterazione di SOR in (6.6).

b) Provare tale routine sulla matrice che si ottiene attraverso il comando `full(gallery('poisson',3))` scegliendo il termine noto b in modo che la soluzione di riferimento sia fatta da tutti 1, con $\omega = 1.1$.

c) Riportare l'andamento sia dell'errore relativo sia del residuo ($||b - Ax^{(k)}||_2$) e confrontarlo con gli andamenti dell'errore relativo e del residuo ottenuti con il metodo di Jacobi e di Gauss-Seidel. (Scala semi-logaritmica)

Suggerimenti:

- A partire dalla `routine` per Gauss-Seidel modificarla appropriatamente seguendo la formula in (6.6), in particolare il metodo SOR deve prendere in ingresso anche il parametro ω.
- I grafici ottenuti devono essere simili a quelli riportati in Figura 6.3.

Esercizio .47. Si consideri la matrice ottenuta mediante il comando `full(gallery('poisson',3))`.

a) Si determini per quale valore di ω il raggio spettrale della matrice d'iterazione B_w in (6.7) è minimo. Si indichi tale valore con $\omega_{\mathbf{ott}}$. Si riporti in un grafico il valore del raggio spettrale in corrispondenza di ω.

b) Si studi la convergenza (tramite un grafico in scala semi-logaritmica) del metodo SOR con $\omega = (\omega_{\mathbf{ott}} - 0.1, \omega_{\mathbf{ott}}, \omega_{\mathbf{ott}} + 0.1)$.

I grafici ottenuti dovrebbero essere come quelli riportati in Figura 6.4.

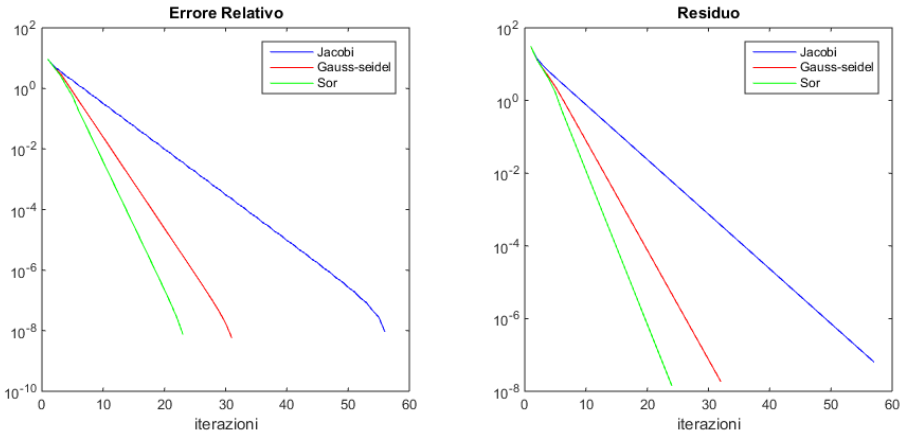

Figura 6.3: Errore relativo e Residuo: confronto.

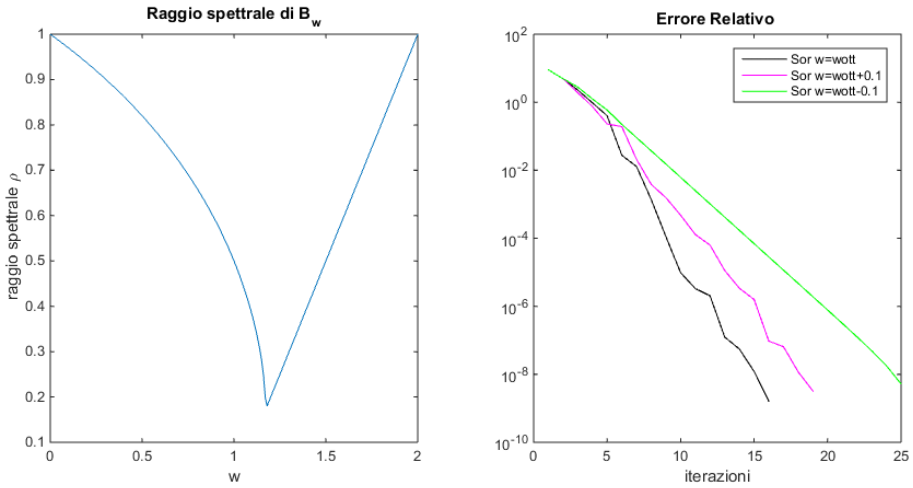

Figura 6.4: Convergenza del metodo SOR al variare di ω.

6.6 Metodo del gradiente

Nei metodi del gradiente si assume la matrice A simmetrica e definita positiva. In questa ipotesi la soluzione del sistema lineare $Ax = b$ è equivalente alla minimizzazione della funzione quadratica $f(x) = \frac{1}{2}x^T Ax - b^T x$. In particolare, questi metodi sono metodi iterativi nei quali al passo $k-$esimo si sceglie una direzione $p^{(k)}$ ed uno scalare $\alpha^{(k)}$ in maniera che, posto

$$x^{(k+1)} = x^{(k)} + \alpha^{(k)} p^{(k)} \tag{6.8}$$

si abbia $f(x^{(k+1)}) < f(x^{(k)})$. In particolare nel metodo del gradiente $p^{(k)}$ è dato dalla direzione del gradiente, i.e.

$$p^{(k)} = -\nabla f(x^{(k)}) = -(Ax^{(k)} - b) = b - Ax^{(k)} = r^{(k)}$$

che risulta essere coincidente con quella del residuo, mentre la costante $\alpha^{(k)}$ è scelta in modo tale da realizzare il minimo della funzione nella direzione $p^{(k)}$, i.e.

$$\alpha^{(k)} = \arg\min_{\alpha \in \mathbb{R}} f(x^{(k)} + \alpha p^{(k)})$$

che corrisponde a scegliere:

$$\alpha^{(k)} = \frac{{p^{(k)}}^T r^{(k)}}{{p^{(k)}}^T A p^{(k)}} = \frac{{r^{(k)}}^T r^{(k)}}{{r^{(k)}}^T A r^{(k)}}.$$

Esercizio .48. Scrivere una `routine` $MATLAB$ che implementi il metodo del gradiente. Provare tale routine scegliendo il termine noto b in modo che la soluzione di riferimento sia fatta da tutti 1 e usando come matrici quelle che si ottengono con i seguenti comandi

```
A1= full(gallery('poisson',3));
A2= full(gallery('tridiag',10));
```

Riportare infine l'andamento dell'errore in funzione del numero di iterazioni.

Suggerimenti per l'implementazione della `routine` per il metodo del gradiente:

- La function deve prendere come argomenti in **input** la matrice del sistema A, il termine noto b, un'approssimazione iniziale della soluzione x_0, la tolleranza con cui si vuole cercare la soluzione `tol`, e il numero massimo di iterazioni che concediamo al metodo `maxiter`.
- La function deve restituire in **output**: la soluzione del sistema (a meno della tolleranza concessa) x, una variabile booleana che indichi se il metodo è arrivato a convergenza oppure no, e una matrice `xhist` sulle cui colonne siano presenti tutte le soluzioni calcolate durante le varie iterazioni (in particolare sulla prima colonna deve esserci x_0 e sull'ultima x).
- Inizializzare la variabile booleana, un contatore delle iterazioni, la matrice `xhist`.
- Costruire un ciclo `while` per il calcolo dell'iterazione descritta in (6.8). Utilizzare un **test d'arresto** basato sul residuo (vedi lezione precedente, sezione 2.4); fare attenzione al caso del vettore b nullo o comunque troppo piccolo.

- Salvare ad ogni iterazione il risultato del prodotto tra A e r in modo da effettuare il prodotto matrice vettore una sola volta per ciascuna iterazione invece che due. (Finita ciascuna iterazione tale quantità non serve più.)
- Terminato il ciclo: occuparsi della variabile booleana e di ridimensionare, se necessario, la matrice `xhist`.
- Per la seconda matrice, si dovrebbe ottenere un grafico come quello riportato in Figura (6.5).

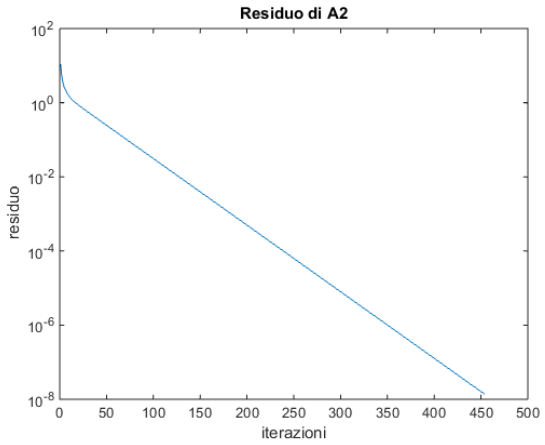

Figura 6.5: Andamento del residuo per la matrice A2 con il metodo del gradiente.

6.7 Metodo del gradiente coniugato

Nel metodo del gradiente coniugato si assume la matrice A simmetrica e definita positiva. In questa ipotesi la soluzione del sistema lineare $Ax = b$ è equivalente alla minimizzazione della funzione quadratica $f(x) = \frac{1}{2}x^T Ax - b^T x$. In particolare, questi metodi sono metodi iterativi nei quali al passo $k-$esimo si sceglie una direzione $p^{(k)}$ ed uno scalare $\alpha^{(k)}$ in maniera che, posto

$$x^{(k+1)} = x^{(k)} + \alpha^{(k)} p^{(k)} \tag{6.9}$$

si abbia $f(x^{(k+1)}) < f(x^{(k)})$. In particolare nel metodo del gradiente coniugato $p^{(k)}$ è scelto in modo da verificare la ricorrenza

$$\begin{aligned} p^{(0)} &= r^{(0)} \quad \text{pari al residuo iniziale} \\ p^{(k+1)} &= r^{(k+1)} - \beta_k p^{(k)} \end{aligned}$$

dove i coefficienti β_k sono scelti in modo da rendere l'insieme dei vettori $p^{(k)}$ A-ortogonali cioè tale da soddisfare la relazione

$$p^{(k+1)^T} A p^{(k)} = 0$$

Sostituendo si ha

$$p^{(k+1)^T} A p^{(k)} = r^{(k+1)^T} A p^{(k)} - \beta_k p^{(k)^T} A p^{(k)}$$

da cui

$$\beta_k = \frac{r^{(k+1)^T} A p^{(k)}}{p^{(k)^T} A p^{(k)}}.$$

La costante α_k è scelta in modo tale da realizzare il minimo della funzione nella direzione $p^{(k)}$, i.e.

$$\alpha^{(k)} = \arg\min_{\alpha \in \mathbb{R}} f(x^{(k)} + \alpha p^{(k)})$$

che corrisponde a scegliere:

$$\alpha_k = \frac{p^{(k)^T} r^{(k)}}{p^{(k)^T} A p^{(k)}}.$$

Esercizio .49. Scrivere una `routine` *MATLAB* che implementi il metodo del gradiente coniugato. Provare tale routine scegliendo il termine noto b in modo che la soluzione di riferimento sia fatta da tutti 1 e usando come matrici quelle dell'esercizio precedente. Riportare infine l'andamento dell'errore in funzione del numero di iterazioni e confrontarlo con i risultati ottenuti col metodo del gradiente.
Commentare i risultati ottenuti.

Suggerimenti per l'implementazione della `routine` per il metodo del gradiente coniugato:

- Modificare opportunamente la routine del metodo del gradiente dell'esercizio precedente.
- Salvare ad ogni iterazione il risultato del prodotto tra A e p in modo da effettuare il prodotto matrice vettore una sola volta per ciascuna iterazione invece che due.
- I risultati ottenuti per la seconda matrice dovrebbero essere come quelli riportati in Figura 6.6.

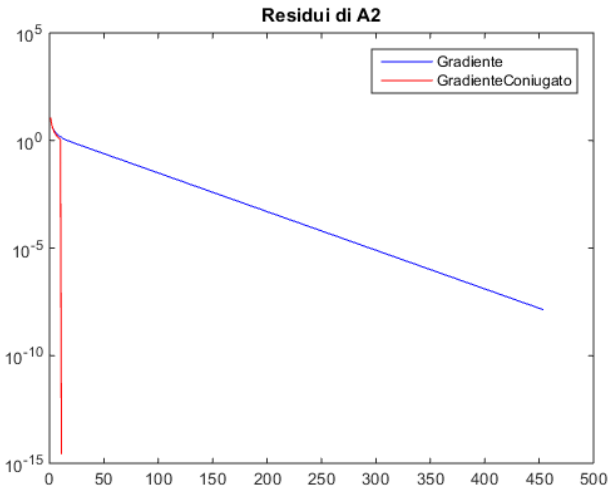

Figura 6.6: Confronto tra metodo del gradiente e del gradiente coniugato.

6.8 Perché e come evitare il calcolo diretto dell'inversa di una matrice

Prima di tutto, consideriamo il caso di un sistema lineare (non singolare)

$$A\mathbf{x} = \mathbf{b}, \quad A \in \mathbb{R}^{n\times n},\ b \in \mathbb{R}^{n\times 1}.$$

In MATLAB il comando

$$\mathbf{x} = A\backslash\mathbf{b};$$

calcola la soluzione del sistema lineare, usando un opportuno metodo diretto. Sarebbe ovviamente possibile usare il comando `inv` per il calcolo dell'inversa A^{-1} e calcolare $\mathbf{x}$ tramite il comando `x = inv(A)*b`. Tuttavia, tale metodo risulta essere svantaggioso, sia dal punto di vista computazionale che dal punto di vista dell'accuratezza. Dal punto di vista dell'accuratezza, si consideri il sistema lineare formato dall'unica equazione

$$49x = 49.$$

L'applicazione del metodo di eliminazione gaussiana porta al comando $\mathbf{x} = 49/49$, mentre l'inversione di matrice porta al comando $x = (1/49) * 49$. É facile verificare che solo nel primo caso si ottiene esattamente $x = 1$. **La regola fondamentale è: mai calcolare l'inversa di una matrice**, a meno che non sia esplicitamente richiesta. Cioè ha senso calcolare l'inversa quando serve esattamente questa matrice, e non tutte le volte che l'oggetto richiesto è la matrice moltiplicata per altro.

Per esempio, dovendo valutare un'espressione del tipo $\boxed{A^{-1}\mathbf{b}}$, si può considerare il sistema $A\mathbf{x} = \mathbf{b}$ la cui soluzione $\mathbf{x}$ è proprio la quantità richiesta $\mathbf{x} = A^{-1}\mathbf{b}$ (per risolvere il sistema si può usare il comando \ di MATLAB).

Inoltre dovendo valutare $\boxed{A^{-1}B}$, dove $B \in R^{n\times n}$ è una matrice, è comunque possibile considerare sistemi lineari $AX = B$ (in cui la colonna i della matrice X è la soluzione del sistema lineare di matrice A e termine noto la colonna i di B). In tal caso il comando \ risolve contemporaneamente gli n sistemi associati e quindi la matrice X ottenuta come

$$X = A\backslash B.$$

è la quantità richiesta. Analogamente, un'espressione del tipo $\boxed{BA^{-1}}$ può essere calcolata con il comando B/A.

Esercizio .50. Nell'esercizio .42 avevamo calcolato le matrici di iterazione di Jacobi e Gauss Seidel attraverso l'uso del comando `inv`. Ripetere l'esercizio sostituendo il calcolo delle matrici inverse con la soluzione di un opportuno sistema lineare.

Esercizio .51. Implementare una variante dell'algoritmo di Jacobi e/o dell'algoritmo di Gauss Seidel utilizzando la versione matriciale dell'algoritmo e testare il funzionamento delle routine su alcuni esempi. (Fare riferimento agli esercizi .43 e .44: in particolare con la versione matriciale sarà necessario solo il ciclo `while` e nessun ciclo `for` interno). Variare il tipo di test d'arresto utilizzato. Evitare il calcolo della matrice inversa (è permesso l'uso del comando \ di Matlab).

6.9 Soluzione di sistemi non lineari con il metodo di Newton

A volte dobbiamo risolvere dei sistemi di equazioni f_i che però *non sono lineari*: in tal caso i metodi studiati fino ad adesso non si possono applicare direttamente.
Vediamo qualche esempio

$$\begin{cases} f_1 : x_1^2 + x_2^2 = 0 \\ f_2 : e^{x_1} + e^{x_2} = log(x_3) \\ f_3 : x_1 x_2 x_3 = x_1 \end{cases} \qquad \begin{cases} f_1 : x_1^2 + x_2^2 = 1 \\ f_2 : \sin(\frac{\pi x_1}{2}) + x_2^3 = 0. \end{cases} \tag{6.10}$$

In generale un sistema non lineare di n funzioni in n incognite, si può scrivere come

$$\begin{cases} f_1(x_1, \ldots, x_n) = 0 \\ f_2(x_1, \ldots, x_n) = 0 \\ \vdots \\ f_n(x_1, \ldots, x_n) = 0 \end{cases} \tag{6.11}$$

dove $f_i : \mathbb{R}^n \to \mathbb{R}, i = 1, \ldots n$ sono funzioni non lineari. Posto $\mathbf{f} = (f_1, \ldots, f_n)^t, \mathbf{x} = (x_1, \ldots, x_n)^t$ può riscriversi compattamente come

$$\mathbf{f}(\mathbf{x}) = \mathbf{0}.$$

Inoltre possiamo indicare la *matrice jacobiana* con

$$\mathbf{J_f}(\mathbf{x}) = \left(\frac{\partial f_i}{\partial x_j}\right) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & & & \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_n} \end{bmatrix}$$

(cioè una matrice che ha sulla riga i il gradiente della funzione f_i).

Se riuscissimo per esempio a scrivere questo sistema di equazioni come un sistema lineare della forma $Ax = b$ poi saremmo in grado di risolverlo.

Richiamiamo il *metodo di Newton* per trovare gli zeri di *una* funzione non lineare $f(x) = 0$.

- Si fissa una approssimazione iniziale della soluzione x_0.
- Si calcola la derivata $f'(x)$ della funzione.
- Si fa un ciclo `while` che si arresta o se viene superato un numero massimo di iterazioni o se la soluzione trovata è abbastanza vicina a quella esatta (vedi criterio dello scarto, lezione precedente, sezione 2.4).
- All'interno del ciclo `while` si cerca una nuova soluzione x^{n+1} a partire dalla vecchia x^n secondo la seguente iterazione di punto fisso

$$x^{n+1} = x^n - \frac{f(x^n)}{f'(x^n)}. \tag{6.12}$$

Il metodo di Newton può essere applicato non solo alle singole equazioni, ma anche ai sistemi di equazioni, basta adottare qualche **accorgimento**:

- Non si può usare una semplice derivata $f'(x)$, ma si deve sostituire con la matrice jacobiana $\mathbf{J_f}(\mathbf{x})$.
- Non si può dividere per una matrice!

  Quindi l'iterazione in (6.12) diventa $\mathbf{J_f}(\mathbf{x}_n)(\mathbf{x}_{n+1} - \mathbf{x}_n) = -\mathbf{f}(x_n)$ e chiamando $\delta = \mathbf{x}_{n+1} - \mathbf{x}_n$

$$\mathbf{J_f}(\mathbf{x}_n)\delta = -\mathbf{f}(x_n). \tag{6.13}$$

- Ciò che bisogna risolvere in (6.13) è un sistema lineare: la matrice è rappresentata dalla matrice jacobiana $\mathbf{J_f}(\mathbf{x}_n)$, il vettore delle incognite è δ e il termine noto è rappresentato da $-\mathbf{f}(x_n)$. Possiamo quindi usare o uno dei metodi studiati o la routine \ di $MATLAB$.
- Visto che δ rappresenta la differenza tra due iterate successive, **il test dello scarto** si può effettuare direttamente su questa variabile.

Esercizio .52. Risolvere i sistemi di equazioni non lineari in (6.10) utilizzando il metodo di Newton descritto qui sopra (cioè il metodo già visto per le equazioni con i dovuti accorgimenti per applicarlo ad un sistema). Commentare in modo preciso e completo ogni fase della soluzione.

Suggerimenti:

- Le equazioni dei sistemi si possono vedere come una sola funzione vettoriale, cioè per esempio il sistema in (6.10) si può scrivere come

```
1     f1 = @(x) [ x(1)^2 + x(2)^2; ...
2     exp(x(1)) + exp(x(2)) - log(x(3)); ...
3     x(1)*x(2)*x(3) - x(1) ];
```

  dove `x` è un vettore di tre componenti. Di conseguenza la matrice jacobiana è una funzione sempre di `x`, e sarà rappresentata da una matrice 3×3.
- Seguire l'idea classica dell'algoritmo (approssimazione iniziale, ciclo `while`) e per calcolare $\mathbf{x}_{n+1}$ usare (6.13) invece di (6.12).
- Per risolvere il sistema lineare in (6.13) usare la routine $MATLAB$ \.
- Per controllare se la soluzione $\mathbf{x}$ trovata è corretta basta verificare se $\mathbf{f}(\mathbf{x})$ è sufficientemente vicino a zero.

Capitolo 7

Calcolo di autovalori e autovettori

7.1 Metodo delle potenze

Il metodo delle potenze è un metodo iterativo per il calcolo approssimato dell'autovalore di modulo massimo di una matrice A diagonalizzabile con autovalori $|\lambda_1|>|\lambda_2| \geq \cdots \geq |\lambda_n|$ ed il corrispondente autovettore.

L'algoritmo si basa sulla seguente idea. Fissato un vettore arbitrario $q_0 \in \mathbb{C}^n$ con norma euclidea unitaria, si genera la seguente successione

$$\begin{aligned} z^{(k)} &= Aq^{(k-1)} \\ \lambda^{(k)} &= q^{(k-1)^H} z^{(k)} \\ q^{(k)} &= z^{(k)}/||z^{(k)}||_2. \end{aligned} \tag{7.1}$$

Si può allora dimostrare che la successione dei $q^{(k)}$ tende all'autovettore di modulo massimo x_1 mentre $\lambda^{(k)}$, tende all'autovalore di modulo massimo λ_1.

Osservazioni relative al metodo delle potenze

1. L'analisi di convergenza mostra che l'efficacia del metodo delle potenze dipende fortemente dal fatto che l'autovalore dominante sia ben separato, cioè $\left|\frac{\lambda_2}{\lambda_1}\right| \ll 1$. Nel caso in cui ci siano due autovalori di modulo massimo uguale, $|\lambda_1| = |\lambda_2|$ abbiamo:

   a) $\lambda_2 = \lambda_1$: i due autovalori dominanti sono coincidenti. In questo caso il metodo è ancora convergente e $\lambda^{(k)} \to \lambda_1$ ma $q^{(k)}$ converge ad un vettore nel sottospazio genrato da x_1 e x_2 (quindi non all'autovettore corretto).

   b) $\lambda_2 = -\lambda_1$: i due autovalori dominanti sono opposti. In questo caso l'autovalore di modulo massimo può essere approssimato applicando il metodo delle potenze alla matrice A^2, in modo che $\lambda_2^2 = \lambda_1^2$ e l'analisi ricade nel punto precedente.

   c) $\lambda_2 = \bar{\lambda}_1$: $[\dots]$.

2. Per quanto riguarda l'implementazione al calcolatore vale la pena notare che le normalizzazioni dei vettori $q^{(k)}$ a 1.0 evita gli errori di overflow (quando $\lambda_1 > 1$) e underflow (quando $\lambda_1 < 1$).

3. Un'ulteriore richiesta è che il vettore iniziale $q^{(0)} = \sum_{k=1}^{n} \alpha_k x_k$ abbia una componente $\alpha_1 \neq 0$ lungo l'autovettore associato all'autovalore di modulo massimo x_1. Infatti, può essere dimostrato che, lavorando in aritmetica esatta, la sequenza $q^{(k)}$ converge alla coppia (λ_2, x_2) se $\alpha_1 = 0$. Nonostante ciò l'insorgere di (inevitabili) errori di arrotondamento assicura che in pratica la successione $q^{(k)}$ contiene una componente non nulla anche nella direzione di x_1 e questo consente al metodo di convergere verso l'autovalore λ_1.

   (Per calcolare il trasposto coniugato si può usare il comando $MATLAB$ `ctranspose`).

Esercizio .53. Scrivere una routine MATLAB che implementi il metodo delle potenze. Fare in modo che l'algoritmo restituisca anche le approssimazioni di lambda calcolate in ciascuna iterazione.

a) Testare l'algoritmo per il calcolo dell'autovalore dominante e del corrispondente autovettore sulle seguenti matrici

```
A1= full(gallery('poisson',3));
A2= full(gallery('tridiag',10)).
```

b) Per quanto riguarda la matrice $A2$ si scelga come approssimazione iniziale il vettore $z_0 = (1, 1, \ldots, 1)$: in tal caso ci si trova nella situazione descritta dall'osservazione 3. Il metodo inizialmente converge a λ_2, quindi bisogna forzarlo a fare più iterazioni (per esempio imponendo una tolleranza negativa o ...) in modo da introdurre degli errori numerici che permettano di arrivare a trovare λ_1.

c) Per verificare i calcoli si disegni l'andamento al variare dell'iterazione dell'errore relativo commesso (come valore di riferimento usare l'autovalore di modulo massimo ottenuto con il comando `eig` di $MATLAB$) e dello scarto tra due iterate successive.

d) Commentare i risultati ottenuti.

Si dovrebbero ottenere grafici simili a quelli riportati in Figura .53.

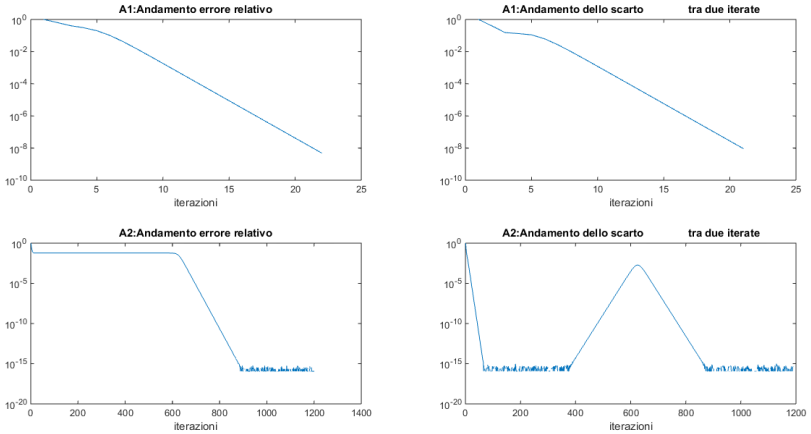

Figura 7.1: I grafici sono stati ottenuti usando $q_0 = [1; 1, \ldots 1]$ sia per A1, sia per A2

7.2 Metodo delle potenze inverse

Applicando il metodo delle potenze alla matrice A^{-1}, si può calcolare l'autovalore minimo λ_n della matrice A. Supponiamo che A sia non singolare, diagonalizzabile e con autovalori tali che:

$$\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_{n-1} > \lambda_n > 0.$$

Come sappiamo, la matrice A^{-1} avrà autovalori $1/\lambda_i$ tali che:

$$\frac{1}{\lambda_n} > \frac{1}{\lambda_{n-1}} \geq \cdots \geq \frac{1}{\lambda_1}.$$

Inoltre A e A^{-1} hanno gli stessi autovettori ($Av = \lambda v \iff A^{-1}v = \lambda^{-1}v$).

Poiché in generale A^{-1} non é nota (e comunque molto costosa da calcolare come visto nella sezione precedente), il vettore $q^{(k+1)}$ si troverà risolvendo il sistema lineare $Aq^{(k+1)} = q^{(k)}$. Tali sistemi si risolvono efficientemente utilizzando la fattorizzazione LU (o di Cholesky) di A.

Note. Il codice per il metodo delle potenze inverse richiede rispetto al metodo delle potenze le seguenti modifiche:

1. Calcolo della fattorizzazione LU (con l'apposito comando $MATLAB$) di A all'esterno del ciclo di convergenza;

2. Sostituzione al prodotto $z = Aq$ con la risoluzione del sistema $Az = q$ mediante uso della fattorizzazione del punto precedente;

3. Calcolo del reciproco di λ. (Perché questo metodo calcola l'autovalore di A^{-1} che è $1/\lambda_n$ e noi vogliamo λ_n)

Esercizio .54. a) Scrivere una routine $MATLAB$ che implementi il metodo delle potenze inverse, seguendo le note qui sopra che spiegano come modificare opportunamente il codice delle potenze. Fare in modo che l'algoritmo restituisca anche le approssimazioni di lambda calcolate in ciascuna iterazione.

b) Usare poi tale algoritmo per calcolare l'autovalore minimo delle matrici dell'esercizio .53. (Come approssimazione iniziale scegliere un vettore casuale generato con il comando `rand` di $MATLAB$).

c) Per verificare i calcoli si disegni l'andamento al variare dell'iterazione dell'errore relativo commesso (l'autovalore minimo di riferimento si può trovare usando il comando `eig` di $MATLAB$).

Si dovrebbero ottenere grafici simili a quelli riportati in Figura 7.2.

7.3 Metodo delle potenze con shifting

Si osservi che da $Ix = x$ e $Ax = \lambda x$ segue che

$$\forall \mu \in \mathbb{C}, \quad (\mu I)x = \mu x \quad \text{e} \quad Bx = (\mu I - A)x = \mu x - \lambda x = (\mu - \lambda)x.$$

Quindi la matrice shiftata $B = (\mu I - A)$ ha autovalori σ_i traslati di μ, i.e. $\sigma_i = \mu - \lambda$ e stessi autovettori di A.

Inoltre si può combinare lo shift con il metodo delle potenze inverse per trovare l'autovalore di A più vicino ad un dato valore µ: $(A - \mu I)q^{(k+1)} = q^{(k)}$.

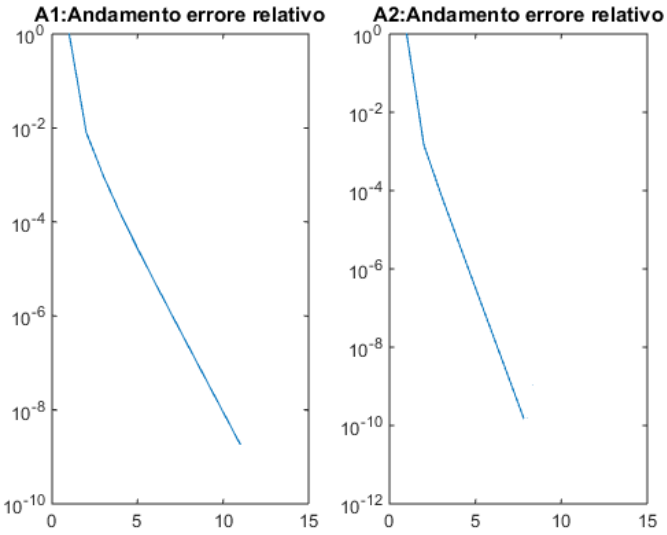

Figura 7.2: Andamento errore relativo per l'autovalore minimo delle matrici dell'esercizio.

Oppure, combinandolo con il metodo delle potenze, si può trovare l'autovalore di A più lontano da μ: $q^{(k+1)} = (A - \mu I)q^{(k)}$.

L'implementazione consiste nelle seguenti modifiche rispetto al metodo delle potenze inverse:

- fattorizzare la matrice $(A - \mu I)$
- una volta calcolato σ (approssimazione dell'autovalore) ricordarsi di calcolare λ come $\lambda = \mu + 1/\sigma$.

Esercizio .55.

a) Scrivere una routine $MATLAB$ che implementi il metodo delle potenze con shifting, seguendo le note qui sopra che spiegano come modificare opportunamente il codice delle potenze inverse. Fare in modo che l'algoritmo restituisca anche le approssimazioni di lambda calcolate in ciascuna iterazione.

b) Testare l'algoritmo su una matrice a scelta e confrontare i risultati ottenuti con quelli dati dal comando `eig`.

Capitolo 8

Interpolazione

Nelle applicazioni si conoscono solitamente dati provenienti da campionamenti di una funzione f sui valori $x_i, i = 0, \ldots, n$, ovvero $(x_i, f(x_i))$ oppure dati sparsi provenienti da misurazioni (x_i, y_i), $i = 0, \ldots, n$. Il problema dell'interpolazione consiste nel trovare una funzione $\tilde{f}$ tale da soddisfare le *condizioni d'interpolazione*

$$\tilde{f}(x_i) = f(x_i), \text{ oppure } \tilde{f}(x_i) = y_i.$$

A seconda della forma di $\tilde{f}$ parleremo di interpolazione polinomiale, interpolazione spline, trigonometrica ecc...

Per prima cosa ci occupiamo dell'interpolazione polinomiale. Esistono vari metodi per costruire il polinomio di interpolazione (che è *unico*), dati punti di interpolazione x_i distinti tra loro: per esempio utilizzando la matrice di *Vandermonde*, i polinomi di Lagrange, o la forma di Newton.

8.1 Matrice di Vandermonde

Il seguente teorema dice che il problema dell'interpolazione polinomiale ha un'unica soluzione se i punti di interpolazione x_i, su cui costruiremo il polinomio interpolante, sono *distinti*.

Teorema 8.1. Dati $n+1$ punti (x_i, y_i), $i = 0, \ldots n$ con $x_i \neq x_j$ se $i \neq j$, esiste un unico polinomio di grado n, $p_n(x) = a_0 + a_1 x + \ldots a_n x^n$ per cui

$$p_n(x_i) = y_i \quad i = 0, \ldots n. \tag{8.1}$$

Le condizioni in (8.1) sono equivalenti al seguente sistema

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^n \\ 1 & x_1 & x_1^2 & \ldots & x_1^n \\ \vdots & & & & \vdots \\ 1 & x_n & x_n^2 & \ldots & x_n^n \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}, \tag{8.2}$$

la cui matrice è detta *matrice di Vandermonde*. Quindi per trovare i coefficienti del polinomio interpolante basta risolvere questo sistema: in generale però non si usa questo metodo perché la matrice di Vandermonde è particolarmente malcondizionata.

Esercizio .56 (Malcondizionamento della matrice di Vandermonde, commentato insieme, 20 min)**.** Prendere n nodi di interpolazione equispaziati nell'intervallo $[-2, 2]$, con n che varia da 2 a 20. Per ciascun n costruire la matrice di Vandermonde e salvarne il numero di condizionamento in un vettore.
Fare un grafico `loglog` che riporti il numero di condizionamento della matrice in corrispondenza della quantità di nodi di interpolazione usati. Com'è la crescita del numero di condizionamento? (cioè è quadratica, cubica, ... , esponenziale o altro ?).

Suggerimenti:

- Per costruire la matrice di Vandermonde esiste il comando `vander`: però restituisce le colonne in ordine diverso da quello in (8.2); per ottenere la matrice usuale usare

```
V = vander(xi);
V = V(:,end:-1:1);    % prende le colonne dall'ultima alla prima
```

  Per calcolare il numero di condizionamento, si può usare il comando `cond`.

- Per capire da un grafico in che modo cresce/decresce una quantità, si possono plottare nella stessa finestra delle curve di riferimento e vedere quale ha la crescita più simile, ad esempio

```
loglog(nnodi, condVander, '*m', nnodi, exp(nnodi), 'k', ...
       nnodi, nnodi.^2, 'b', nnodi, nnodi.^6, 'c')
legend('condizionamento','esponenziale','quadratico', 'alla sesta')
xlabel('numero di nodi')
ylabel('condizionamento')
```

- Il grafico ottenuto dovrebbe essere simile a quello riportato in Figura 8.1.

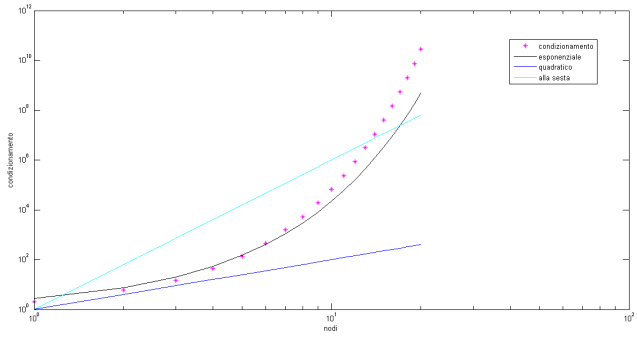

Figura 8.1: Condizionamento della matrice di Vandermonde

8.2 Polinomi di Lagrange

Al posto del sistema con la matrice di Vandermonde si può trovare il polinomio interpolante sfruttando le proprietà dei polinomi di Lagrange, che costituiscono una base meglio condizionata dello spazio dei polinomi.

8.2.1 Definizione e proprietà dei polinomi di Lagrange

Definizione 8.1. I **polinomi elementari di Lagrange** sono definibili a partire dai punti d'interpolazione x_i come segue

$$\ell_i(x) = \prod_{j=0, j\neq i}^{n} \frac{x - x_j}{x_i - x_j}. \tag{8.3}$$

I polinomi $\ell_i, i = 0, \ldots, n$ sono polinomi di grado n, valgono 1 quando $x = x_i$ e 0 quando $x = x_j, j \neq i$, cioè

$$\ell_i(x_j) = \delta_{i,j}. \tag{8.4}$$

Pertanto possiamo esprimere il polinomio d'interpolazione $p_n(x)$, costruito sull'insieme $(x_i, y_i), i = 0 \ldots n$, come

$$p_n(x) = \sum_{i=0}^{n} \ell_i(x) y_i. \tag{8.5}$$

Inoltre vale la seguente identità

$$\sum_{i=0}^{n} \ell_i(x) = 1, \quad \forall x \tag{8.6}$$

cioè i polinomi di Lagrange formano una *partizione dell'unità*.

Esercizio .57 (Proprietà dei polinomi di Lagrange, 30 min)**.**

a) Scegliere $n + 1$ punti di interpolazione (per esempio $n + 1$ punti equispaziati nell'intervallo $[0, 1]$) e costruire i corrispondenti polinomi di Lagrange di grado n, mediante la formula in (8.3). Provare per $n = 2, 3, 4$.
Nei suggerimenti sono proposti 2 algoritmi differenti per la costruzione dei polinomi: il primo è più semplice, mentre il secondo permette di evitare un ciclo `for`. Capire entrambi.

b) Rappresentare graficamente i polinomi.

c) Verificare, guardando i grafici che la proprietà $\ell_i(x_j) = \delta_{i,j}$ è valida.

d) Verificare che anche la proprietà in (8.6) è valida (vedere anche il penultimo suggerimento)

Suggerimenti:

- **1' Metodo.** Creiamo una funzione che costruisca l'i-esimo polinomio di Lagrange e lo valuti su un punto xVal. Poi la invochiamo in modo da poter plottare gli $n + 1$ polinomi su tutti i punti di valutazione che sceglieremo.

  La funzione si costruisce così:

```
function l = polLagrange(xi, xVal, i)
 Costruisce l'i-esimo polinomio di Lagrange sui nodi di interpolazione
 contenuti nel vettore xi e lo valuta sul punto xVal

 INPUT:
 xi -> vettore dei punti di interpolazione
 xVal -> punto su cui valutare il polinomio
 i -> indice del polinomio da costruire
```

```
9
10      OUTPUT:
11      l -> valore dell'i-esimo polinomio nel punto xVal
12
13      xj = setdiff(xi,xi(i));         % commentare il comando: .........
14      l = prod(xVal-xj)/prod(xi(i)-xj);
```

Invochiamo così la funzione per fare i grafici:

```
1       y = zeros(length(xVal),ngrad+1);
2       for i = 1:ngrad+1
3           for j = 1:length(xVal)
4               y(j,i) = polLagrange(xi,xVal(j),i); % commentare........
5           end
6       end
7       plot(xVal,y)     % commentare..................
```

- **2' Metodo**. La funzione `polLagrange` del metodo **1** accetta in input un solo punto di valutazione per volta. Quindi per valutare il polinomio su un intero intervallo è stato necessario introdurre un ciclo. Per evitarlo possiamo usare la seguente funzione

```
1       function l = polLagrange2(xi, xVal, i)
2        Costtruisce l'i-esimo polinomio di Lagrange sui nodi di interpolazione
3        contenuti nel vettore xi e lo valuta sul punto xVal
4        INPUT:
5        xi -> vettore dei punti di interpolazione
6        xval -> vettore (COLONNA!) dei punti su cui valutare il polinomio
7        i -> indice del polinomio da costruire
8        OUTPUT:
9        l -> valori dell'i-esimo polinomio nei punti contenuti nel vettore xVal
10
11      n = length(xi);
12      m = length(xVal);
13
14      l = prod( repmat(xVal,1,n-1)-repmat(xi([1:i-1,i+1:n]),m,1) ,2)/ ...
15          prod( xi(i)-xi([1:i-1,i+1:n]) );   % provare a capire e commentare ...
```

- Per verificare la proprietà in (8.6) si può usare

```
1       plot(xVal, sum(y,2)) % commentare ................
```

- Per fare ordine tra i grafici si può usare il comando `subplot`, e mettere i titoli ai grafici.

Esercizio .58 (Interpolazione mediante polinomi di Lagrange, 15 min)**.** Utilizzare la `function polLagrange2` per valutare graficamente il polinomio interpolante di Lagrange $p_3(x)$ nei punti:

x_i	5	-7	-6	0
y_i	1	-23	-54	-954

Suggerimento: Invocare la `function polLagrange2` per costruire i 4 polinomi di Lagrange, direttamente valutati sull'intervallo di interesse, poi usare la formula in (8.5) per costruire il polinomio interpolante. Il grafico del polinomio è riportato in Figura 8.2.

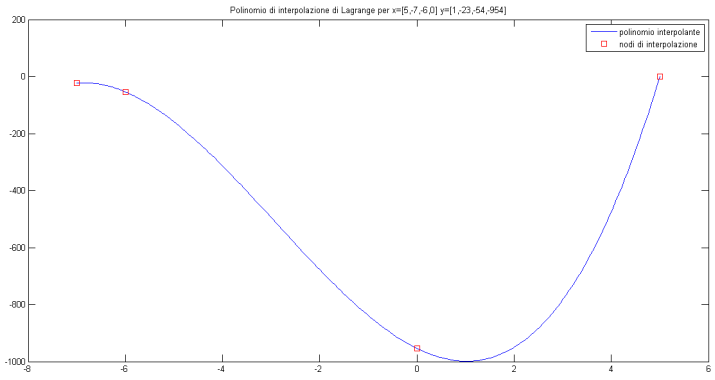

Figura 8.2: Polinomio interpolante per i dati dell'esercizio .58.

8.3 Comando di MATLAB: polyfit

Avendo a disposizione $MATLAB$ si possono usare alcuni comandi predefiniti per calcolare il polinomio interpolante e valutarlo sull'intervallo desiderato, senza dover costruire a mano la matrice di Vandermonde o i polinomi di Lagrange.

La routine $MATLAB$ che implementa l'interpolazione polinomiale è `polyfit`, mediante la sintassi `p = polyfit(x,y,n)` dove n è il grado del polinomio con cui vogliamo interpolare i dati; restituisce in uscita un vettore con i coefficienti del polinomio. Nel nostro caso sarà `n=length(x)-1`. I coefficienti del polinomio p sono da interpretarsi in ordine decrescente: il primo coefficiente è relativo al monomio di grado massimo.
Per valutare un polinomio su un insieme di punti si utilizza la routine `polyval`. (Vedere l'`help` per maggiori dettagli).

Esercizio .59 (Comandi Matlab per interpolazione, autonomamente)**.** Considerando la tabella di punti di interpolazione (x_i, y_i) riportata nell'Esercizio .58, valutare graficamente il polinomio interpolante calcolato attraverso le routine $MATLAB$ `polyfit`. (Con valutare graficamente intendo che dopo averne calcolato i coefficienti, deve essere anche riportato il grafico del polinomio, usando il comando `polyval`).

8.4 Il fenomeno di Runge

Se si ha una funzione $f(x)$ continua in un intervallo $[a, b]$ ed in tale intervallo si calcolano polinomi di interpolazione di grado via via maggiore sembrerebbe naturale aspettarsi che la successione di tali polinomi converga uniformemente ad $f(x)$ in $[a, b]$, ovvero

$$\lim_{n \to +\infty} ||f - P_n||_\infty = 0.$$

Ma nella realtà, per la maggior parte delle funzioni continue, ciò non è vero. Un esempio è fornito dalla *funzione di Runge*

$$f(x) = \frac{1}{1 + x^2} \tag{8.7}$$

nell'intervallo $[-5, 5]$.

Esercizio .60 (Fenomeno di Runge)**.** Costruire il polinomio interpolante, usando $n = 6 : 2 : 12$ nodi di interpolazione ottenuti valutando la funzione di Runge (8.7) nell'intervallo $[-5, 5]$. (Usare i comandi `polyfit` e `polyval`, descritti nella Sezione 8.3, per costruire e graficare il polinomio interpolante).
Riportare il grafico della funzione e dei polinomi interpolanti per alcuni valori di n.
Calcolare la norma infinito della differenza tra il polinomio interpolante valutato e la funzione di Runge e commentare.
I grafici ottenuti dovrebbero essere come quelli riportati in Figura 8.3.
VARIANTI:
a) Si può provare anche ad utilizzare la funzione di Horner vista nella Sezione 3.7.2 al posto del comando `polyval`.
b) Si possono usare i polinomi di Lagrange come nell'esercizio precedente per trovare il polinomio interpolate (al posto di polyfit).
c) Si possono combinare la funzione di Horner e i polinomi di Lagrange (scopo: evitare sempre di calcolare $p_n(x)$ direttamente e usare sempre horner).

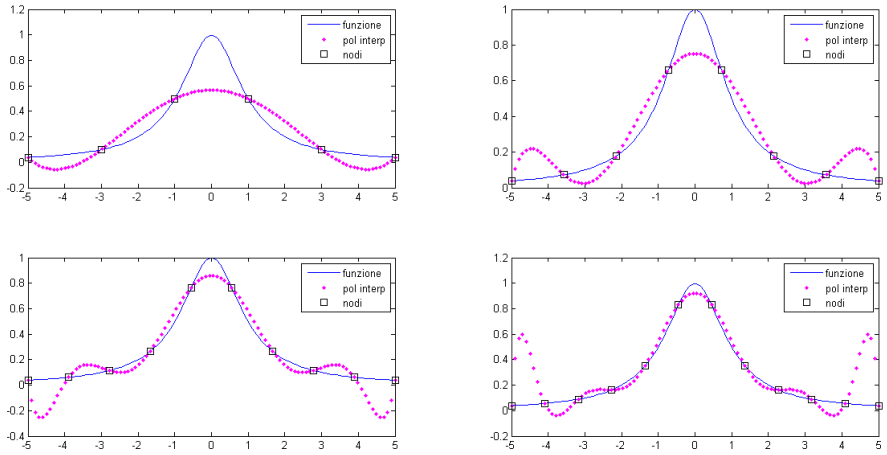

Figura 8.3: Interpolazione della funzione di Runge con nodi equispaziati, con n = 6:2:12.

Dai grafici ottenuti nell'Esercizio .60, possiamo verificare che il polinomio interpolante ha ampie oscillazioni attorno agli estremi. Pertanto non è consigliabile usare punti equidistanti per interpolare $f(x)$. Una scelta ottimale si fa considerando i nodi di *Chebyshev*.

Gli $n+1$ nodi di Chebyshev-Gauss-Lobatto (scalati) sull'intervallo $[a, b]$ sono definiti nel seguente modo

$$x_k = \frac{a+b}{2} - \frac{b-a}{2} \cos\left(\frac{k\pi}{n}\right) \quad \text{per } k = 0, \ldots n.$$

Esercizio .61 (Nodi di Chebyshev)**.** Implementare una `function` che calcola i nodi di Chebyshev per un intervallo qualsiasi.
Ripetere l'Esercizio 5, cercando i polinomi interpolanti costruiti a partire da questi nodi. Riportare i grafici e calcolare la norma infinito dell'errore. Commentare.

I grafici ottenuti dovrebbero essere simili a quelli riportati in Figura 8.4.

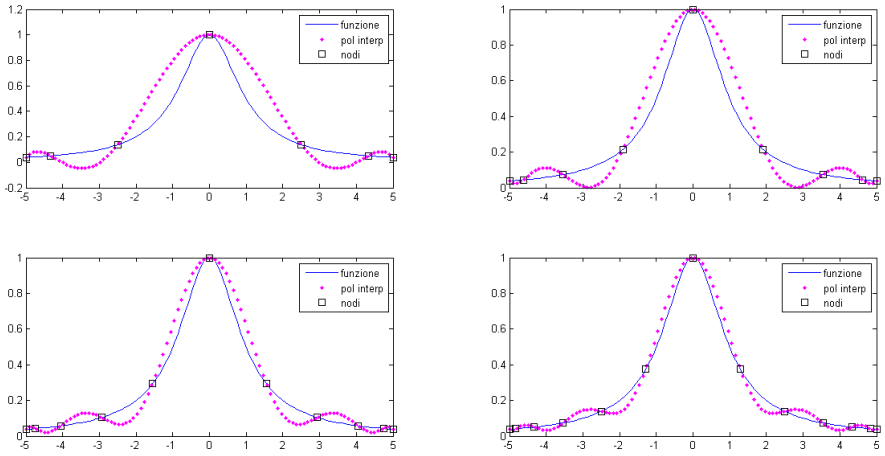

Figura 8.4: Polinomi interpolanti per la funzione di runge su nodi di Chebyshev, con n = 7:2:13 nodi

8.5 Polinomio interpolante in forma di Newton

Si può esprimere il polinomio di interpolazione di grado n della funzione f, in forma di Newton

$$p_n = b_0 + b_1(x - x_0) + b_2(x - x_0)(x - x_1) + \cdots + b_n(x - x_0)\dots(x - x_{n-1}) \qquad (8.8)$$

dove b_i rappresenta la *differenza divisa* di ordine i della funzione f.
Dati $n + 1$ punti distinti $x_0, \dots, x_n$ e i valori $y_i = f(x_i)$

- la *differenza divisa* di ordine 0 della funzione f è $f[x_i] = f(x_i)$,
- la *differenza divisa* di ordine 1 è $f[x_i, x_{i+1}] = \frac{f[x_{i+1}] - f[x_i]}{x_{i+1} - x_i}$
- ricorsivamente, la *differenza divisa* di ordine k è

$$f[x_i, \dots, x_{i+k}] = \frac{f[x_{i+1}, \dots, x_{i+k}] - f[x_i, \dots, x_{i+k-1}]}{x_{i+k} - x_i}.$$

Una `function` che implementa l'algoritmo di calcolo delle differenze divise è il seguente

```
function b = diffDivise(x,y)
% Algoritmo delle differenze divise
% INPUT
% x: vettore dei punti di interpolazione
% y: vettore dei valori nei punti di interpolazione
% OUTPUT
% b: vettore delle differenze divise b = [b_1, ... b_n]

n = length(x);
b = y;

for i = 2:n
    for j = 2:i
        b(i) = (b(i)-b(j-1))/(x(i)-x(j-1));
    end
end
```

Inoltre i b_i nella formula (8.8) sono le differenze divise di ordine i calcolate rispettivamente sui punti

$$\begin{aligned} b_0 &= f[x_0], \\ b_1 &= f[x_0, x_1], \\ &\vdots \\ b_n &= f[x_0, ..., x_n]. \end{aligned}$$

Esercizio .62 (Forma di Newton)**.**

a) Trovare il polinomio che interpola i punti $(0,0), (3,14), (2,5), (1,1)$ facendo uso della forma di Newton in (8.8), e della `function` per le differenze divise riportata qui sopra. Tracciarne un grafico su dei punti di valutazione e riportare nello stesso grafico i nodi di interpolazione.

   Suggerimento: una volta calcolato il vettore delle differenze divise, per implementare la formula in (8.8) si possono usare 2 cicli `for` annidati.

   Si ottiene un grafico come quello riportato Figura 8.5.

b) Mostrare che la seguente proprietà è verificata su un esempio e commentare:
   La differenza divisa di ordine $k+1$ di un polinomio di grado k è identicamente nulla.

c) Mostrare che la seguente proprietà è verificata su un esempio e commentare:
   Le differenze divise sono invarianti rispetto all'ordine dei nodi.

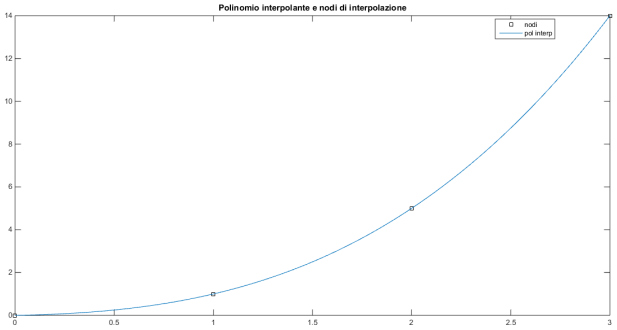

Figura 8.5: Polinomio interpolante

Esercizio .63 (Esercizio riassuntivo)**.** Si consideri la funzione $f(x) = \cos((x^3)/3)(x - 2\pi)e^{-x}$, $x \in [0, \pi]$. Sperimentalmente si determini il grado del polinomio d'interpolazione, costruito sui nodi di Chebyshev in $[0, \pi]$, che approssima $f(x)$ in norma infinito a meno di tol $= 1e-4$. Si scelga il metodo che si preferisce per determinare il polinomio interpolante. Oppure provare con tutti i metodi conosciuti.

Risultati: serve un polinomio di grado 18, il grafico è riportato in Figura 8.6. (Se si usa `polyfit` è normale che appaiano delle warning riguardanti la possibile inaccuratezza del comando se usato con molti nodi, molti cioè più di una decina).

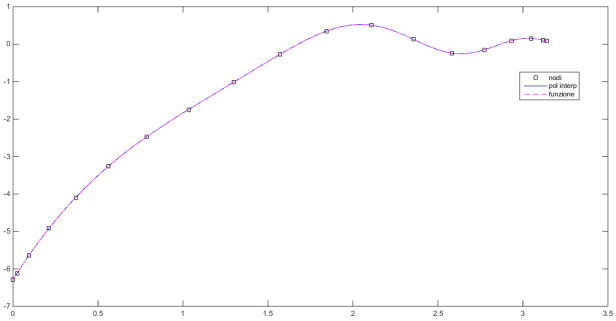

Figura 8.6: Polinomio interpolante dell'Esercizio 8.

8.6 Errore d'interpolazione

Data una funzione f e il suo polinomio interpolante p_n di grado n, costruito a partire dai nodi di interpolazione $x_0, x_1, \dots x_n$, si può dimostrare che l'*errore d'interpolazione* che viene commesso è pari a

$$E_n(x) = \frac{f^{(n+1)}(c)}{(n+1)!} \prod_{j=0}^{n} (x - x_j),$$

dove c è un valore compreso fra i nodi di interpolazione.

Se si vuole stimare l'errore d'interpolazione commesso su un intervallo $[a, b]$ contenente i nodi di interpolazione si può usare la seguente formula

$$\max_{a \le x \le b} |E_n(x)| \le \frac{1}{(n+1)!} \cdot \max_{a \le x \le b} \left| f^{(n+1)}(x) \right| \cdot \max_{a \le x \le b} \left| \prod_{j=0}^{n} (x - x_j) \right|. \tag{8.9}$$

Si può dimostrare inoltre che scegliendo come nodi di interpolazone i punti di Chebyshev sull'intervallo $[-1, 1]$ si ha

$$\max_{-1 \le x \le 1} |E_n(x)| \le \frac{1}{2^n (n+1)!} \cdot \max_{-1 \le x \le 1} \left| f^{(n+1)}(x) \right|. \tag{8.10}$$

Esercizio .64 (Errore di interpolazione)**.** Si consideri la funzione

$$f(x) = \log(2 + x), \quad x \in [-1, 1].$$

Indichiamo con p_n il polinomio di interpolazione di grado n costruito usando i punti di Chebyshev. Sotto tale ipotesi è noto che l'errore di interpolazione si può maggiorare usando la (8.10).

Si considerino i casi $n = 3, 4$ e

- Si faccia un grafico della funzione e del polinomio interpolante (per trovare il polinomio interpolante si usi il metodo che si preferisce).
- Si calcoli il valore fornito dalla (8.10) per la stima dell'errore e si controlli che esso viene rispettato. (Cioè l'errore numerico commesso è $\le$ di quello stimato).

  Suggerimento: per il calcolo della derivata si può usare il comando `diff`, preceduto da `syms x`. Il polinomio interpolante per il caso $n = 4$ dovrebbe apparire come in Figura 8.7.

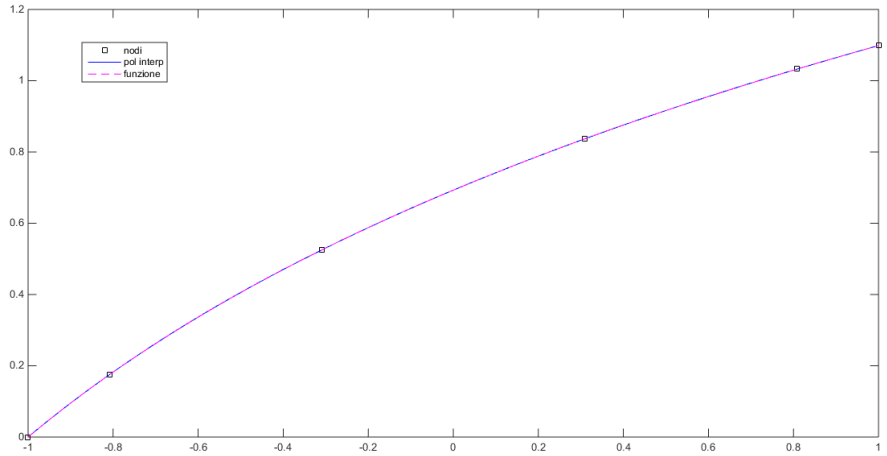

Figura 8.7: Esercizio .64, caso n=4.

Capitolo 9

Integrazione o quadratura numerica

Spesso è necessario calcolare quantità del tipo

$$I(f) = \int_a^b f(x)dx.$$

Questo calcolo può essere eseguito in modo analitico solo se si conosce la primitiva di f e spesso ne vale la pena solo se la primitiva ha un'espressione semplice.
Se invece si verifica una delle seguenti situazioni:

- Non si conosce la primitiva di f,
- La primitiva è una funzione la cui valutazione è troppo costosa,
- La funzione f è nota solo perché si conoscono i valori che essa assume in certi punti, ma non si dispone della sua formulazione analitica,

si rende necessario usare delle formule di integrazione (o quadratura) numerica.

In pratica una formula di quadratura è un'approssimazione dell'integrale che fa uso dei valori della funzione in alcuni punti $x_i, i = 1, \dots m+1$

$$I(f) = \int_a^b f(x)dx \approx \sum_{i=1}^{m+1} \omega_i f(x_i)$$

dove gli x_i sono detti *nodi di quadratura* e i coefficienti ω_i sono detti *pesi* della formula di quadratura.

Osservazione. Ciascuna delle formule che vedremo è inoltre esatta per il calcolo dell'integrale quando l'integranda è un *polinomio* fino ad un certo grado. In tal caso quindi si ottiene lo stesso risultato usando l'integrazione numerica o facendo il calcolo della primitiva, ma spesso si sceglie di usare l'integrazione numerica per rendere più generale (e di più rapida scrittura) il codice (Esempio: metodo degli elementi finiti).

Definizione 9.1. Una formula di quadratura (di tipo interpolatorio) si dice *esatta* con ordine di esattezza n se integra esattamente i polinomi di grado n.

9.1 Il metodo dei trapezi

L'idea alla base del metodo dei trapezi è quella di calcolare l'area sotto la curva $f(x)$ nell'intervallo $[a, b]$ approssimandola con l'area di un trapezio che ha (vedi anche Figura 9.1)

- per *base minore*: $f(a)$,
- per *base maggiore*: $f(b)$,
- per *altezza*: l'intervallo $[a, b]$, $b - a$.

In tal caso si ha

$$I(f) = \int_a^b f(x)\,dx \approx \frac{b-a}{2}\,(f(a) + f(b)).$$

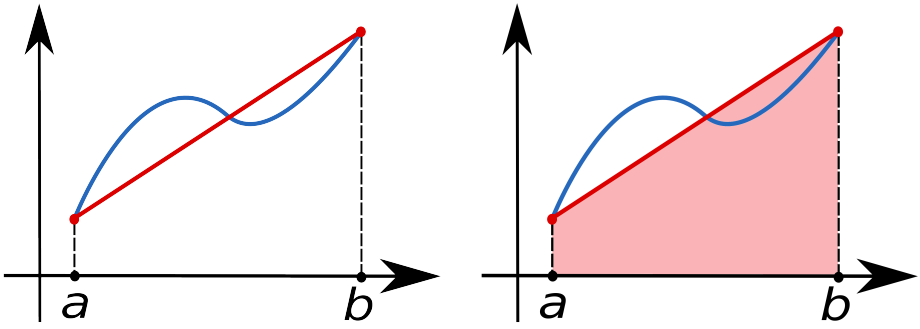

Figura 9.1: Regola del trapezio: permette di calcolare l'area in rosa che approssima l'area sotto la curva.

9.1.1 Formula dei trapezi composita

Per rendere l'approssimazione più accurata si può suddividere l'intervallo $[a, b]$ in m sottointervalli $[x_i, x_{i+1}]$ e applicare la regola dei trapezi in ciascuno di essi (vedi Figura 9.2). (Attenzione alla notazione: m sottointervalli, $m + 1$ nodi di quadratura).

Pertanto si ha

$$I(f) = \int_a^b f(x)\,dx = \sum_{i=1}^{m} \int_{x_i}^{x_{i+1}} f(x)\,dx \approx \sum_{i=1}^{m} \frac{x_{i+1} - x_i}{2}\,(f(x_i) + f(x_{i+1})), \tag{9.1}$$

e nel caso in cui gli intervalli siano uguali, posto $h = (b - a)/m$ si ha

$$x_1 = a,\ x_2 = a + h,\ \ldots,\quad x_i = a + (i-1)h,\quad \ldots,\ x_{m+1} = a + mh,$$

e si ottiene

$$\begin{aligned} I(f) &\approx \frac{h}{2}\left(f(a) + 2\sum_{i=1}^{m-1} f(a + ih) + f(b)\right) = h\left(\frac{f(a)}{2} + \sum_{i=1}^{m-1} f(a + ih) + \frac{f(b)}{2}\right) \\ &= h\left(\frac{f(x_1)}{2} + \sum_{i=2}^{m} f(x_i) + \frac{f(x_{m+1})}{2}\right) \end{aligned} \tag{9.2}$$

(per ottenere tale formula notare che nella somma in (9.1) $f(x_i)$ è ripetuto 2 volte per ogni $i = 2, \ldots, m$, cioè non sono ripetuti due volte solo $f(a)$ e $f(b)$).

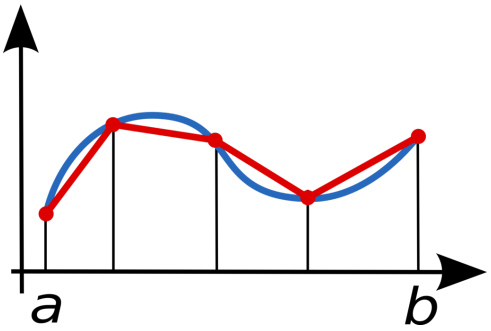

Figura 9.2: Formula dei trapezi composita.

Per quanto riguarda l'**errore** è possibile dimostrare che

$$E_n(f) = \frac{(b-a)^3}{12n^2} f''(\xi), \quad \xi \in (a,b) \tag{9.3}$$

quindi la formula è esatta per polinomi di grado 1 e l'errore decresce con ordine 2. Inoltre se $f''(x) > 0$, cioè la funzione è convessa, si avrà un'approssimazione per eccesso del valore dell'integrale. Invece se $f''(x) < 0$, cioè la funzione è concava, si avrà un'approssimazione per difetto del valore dell'integrale.

9.2 Formula di Cavalieri-Simpson

Si considerino m sotto-intervalli dell'intervallo $[a,b]$, sia m **pari**. Vale la seguente formula

$$\begin{aligned} I(f) &= \sum_{i=1}^{m/2} \int_{x_{2i-1}}^{x_{2i+1}} f(x)\,dx \simeq \frac{h}{3} \sum_{i=1}^{m/2} \Big(f(x_{2i-1}) + 4f(x_{2i}) + f(x_{2i+1}) \Big) = \\ &= \frac{h}{3} \left(f(a) + 4 \sum_{i=2, i \text{ pari}}^{m} f(a+(i-1)h) + 2 \sum_{i=3, i \text{ dispari}}^{m-1} f(a+(i-1)h) + f(b) \right) = \\ &= \frac{h}{3} \left(f(x_1) + 4 \sum_{i=2, i \text{ pari}}^{m} f(x_i) + 2 \sum_{i=3, i \text{ dispari}}^{m-1} f(x_i) + f(x_{m+1}) \right) \end{aligned} \tag{9.4}$$

Per quanto riguarda l'**errore** è possibile dimostrare che

$$E_n(f) = -\frac{(b-a)^5}{180n^4} f^{(4)}(\xi), \quad \xi \in (a,b) \tag{9.5}$$

quindi la formula è esatta per polinomi fino al grado 3 e l'errore decresce con ordine 4.

9.3 Esercizi

Esercizio 8.1 Trapezi

Si scriva una routine $MATLAB$ che calcoli l'integrale di una funzione f nell'intervallo $[a,b]$ usando la formula dei trapezi composita in (9.2).

a) Si testi la routine per il calcolo dell'integrale di un polinomio di grado 1. Come si capisce facendo riferimento alla formula in (9.3) il valore ottenuto con tale formula su un polinomio di primo grado (cioè con derivata seconda nulla) deve essere esatto (a meno della precisione macchina). Confrontare quindi il valore ottenuto con quello esatto (calcolato a mano). Commentare.

b) Si approssimi poi il valore di

$$\int_1^2 \left(e^x + \frac{10}{x^2}\right) dx.$$

c) Confrontare il valore ottenuto con il valore che $MATLAB$ calcola mediante una sua apposita function (se non si conosce il nome di un comando $MATLAB$ si può usare `lookfor` seguito da una parola chiave, in inglese, per cercare funzioni adatte).

d) Si tracci un grafico che mostri l'andamento dell'errore al crescere del numero di sottointervalli in cui si divide $[a, b]$. Rispetta le attese? Commentare. (**N.B.** Con errore NON intendo l'errore delle formule, MA l'errore numerico commesso usando la function, rispetto ad un valore di riferimento; Le attese riguardano l'ordine di convergenza e l'esattezza della formula in certi casi, vedi commenti fatti sotto la formula dell'errore.)

Esercizio 8.2 Cavalieri Simpson

Si scriva una routine $MATLAB$ che calcoli l'integrale di una funzione f nell'intervallo $[a, b]$ usando la formula di Cavalieri-Simpson in (9.2).

a) Si testi la routine per il calcolo dell'integrale di un polinomio di grado 3. Come si capisce facendo riferimento alla formula in (9.5) il valore ottenuto con tale formula su un polinomio di grado ≤ 3 (cioè con derivata quarta nulla) deve essere esatto (a meno della precisione macchina). Confrontare quindi il valore ottenuto con quello esatto. Commentare.

b) Si approssimi poi il valore di $\int_1^2 \left(e^x + \frac{10}{x^2}\right) dx$.

c) Confrontare il valore ottenuto con il valore che $MATLAB$ calcola mediante una sua apposita function.

d) Si tracci un grafico che mostri l'andamento dell'errore al crescere del numero di sottointervalli in cui si divide $[a, b]$. Rispetta le attese? Commentare.

e) Si determini empiricamente/numericamente quale è il numero minimo di sotto-intervalli in cui deve essere suddiviso l'intervallo di integrazione affinchè l'errore sia inferiore a $1e-4$. (Cioè calcolare per diversi m il valore approssimato e confrontare con il valore esatto; se avessi chiesto di stabilirlo teoricamente allora si sarebbe dovuto sfruttare la formula per l'errore).

Esercizio 8.3 Quadratura per un insieme di dati equispaziati

È fornito il seguente insieme di dati sparsi (le x sono equispaziate)

$$x = [0,\ 0.5236,\ 1.0472,\ 1.5708,\ 2.0944,\ 2.6180,\ 3.1416];$$
$$y = [0,\ 0.8440,\ 2.4679,\ 4.8105,\ 7.0326,\ 6.8541,\ 0].$$

Utilizzare il metodo dei trapezi e/o il metodo di Cavalieri Simpson per stimare l'area compresa tra l'asse x e l'interpolante di questi dati.

Risultati: se si usa il metodo dei trapezi si dovrebbe ottenere $I \approx 11.5239$, se si usa il metodo di Cavalieri si ottiene $I \approx 12.0489$.

Esercizio 8.4 Quadratura punti non equispaziati

Implementare il metodo di trapezi e/o di Cavalieri Simpson per un insieme di punti non equispaziati. Testare il proprio metodo su una funzione a scelta su un insieme di punti non equispaziati.

L'esame...

L'esame per la parte di laboratorio consisterà nella soluzione di 3 esercizi attraverso l'implementazione di codice in MATLAB e la motivazione dei risultati ottenuti.

Per ciascun esercizio sarà necessario scrivere un file principale che eventualmente ne invocherà altri. I **file principali** di ciascun esercizio devono chiamarsi rispettivamente `main1.m, main2.m, main3.m`. All'interno dei file principali deve essere riportato il vostro **nome, cognome, numero di matricola**.

Al termine dell'esame dovete inviare una **mail dal vostro indirizzo istituzionale** agli indirizzi **elena.gaburro@unitn.it** e **leonardpeter.bos@univr.it** con:
OGGETTO: Esame calcolo numerico
ALLEGATO: un **unico file zip** (con nome così composto: Nome_Cognome_Matricola, es. Mario_Rossi_VR123456) contenente **tutti** i file destinati alla correzione.
Chi volesse **ritirarsi** deve comunque mandare una mail con nome, cognome, numero di matricola e il testo "MI RITIRO" (se si vuole mantenere un voto ottenuto in un appello precedente ciò deve essere segnalato esplicitamente nella mail).

Chi consegna l'esame **annulla automaticamente** qualsiasi voto ottenuto negli appelli precedenti.

È concesso l'uso di appunti, dispense e dei propri codici.
È vietato l'uso di cellulari ed è vietato comunicare con i compagni in qualsiasi modo.

Le immagini devono essere chiaramente interpretabili.
Il codice deve essere ordinato, indentato e ogni passaggio deve essere adeguatamente motivato.

È obbligatorio **riconsegnare il testo dell'esame** al termine della prova.

Per prepararsi all'esame

Per prepararsi all'esame oltre a svolgere *tutti* gli esercizi (in particolare implementando tutte le function corrispondenti ad algoritmi classici) e ad aver chiara la teoria, consiglio di riflettere sui seguenti punti.

- Criteri d'arresto. Quindi criteri basati sullo scarto tra due iterate successive o sul residuo nel caso dei sistemi lineari . . . , criteri basati sul numero massimo di iterazioni, oppure con uso di tolleranze assolute, relative, o un criterio d'arresto misto . . . applicati a quantità scalari o vettoriali. Operatori logici.

- Ordine di convergenza dei metodi. Cioè per esempio se esiste una formula per l'errore saper dedurre cosa ci si aspetta dal metodo nelle varie situazioni. Quali sono i casi in cui il metodo (pur essendo implementato correttamente) potrebbe aver un ordine differente (migliore o peggiore). Che modifiche si possono applicare al metodo per ripristinare l'ordine di convergenza atteso .... Come capire l'ordine di convergenza o da un grafico o attraverso certi calcoli.
- Grafici. Quali sono i più appropriati alle varie situazioni. Come si ricavano informazioni dai grafici (confronto con curve di crescita caratteristiche).
- Condizioni in cui i metodi possono essere applicati. Per esempio come devono essere le matrici affinché un certo metodo possa essere usato, come devono essere le funzioni (sia quelle direttamente da studiare, che quelle usate nei metodi) ...
- Capire bene il meccanismo alla base della soluzione di sistemi lineari mediante decomposizione di matrici.
- Capire bene come implementare un metodo iterativo qualsiasi.
- Saper inserire in MATLAB una matrice: sia per inserimento diretto di ogni sua componente, sia usando il camando `diag` per matrici diagonali, tridiagonali ..., sia mediante concatenazione/somma/sottrazione di matrici.

L'esame sarà composto da tre esercizi: riporto un esempio di un possibile esercizio strutturato come all'esame.

Esercizio 1

Data una matrice A tridiagonale, essa può essere decomposta nel prodotto di una matrice L e una matrice U ($A = LU$) costruite come segue

$$A = \begin{pmatrix} d_1 & u_1 & & & \\ \ell_1 & d_2 & u_2 & & \\ & \ddots & \ddots & \ddots & \\ & & & & u_{N-1} \\ & & & \ell_{N-1} & d_N \end{pmatrix}, \; L = \begin{pmatrix} 1 & & & & \\ \beta_1 & 1 & & & \\ & \beta_1 & 1 & & \\ & & \ddots & \ddots & \\ & & & \beta_{N-1} & 1 \end{pmatrix}, \; U = \begin{pmatrix} \alpha_1 & u_1 & & & \\ & \alpha_2 & u_2 & & \\ & & \ddots & \ddots & \\ & & & \alpha_{N-1} & u_{N-1} \\ & & & & \alpha_N \end{pmatrix}$$

con

$$\begin{aligned} &\alpha_1 = d_1 \\ &\forall i \in [1, \ldots, N-1] : \quad \beta_i = \frac{\ell_i}{\alpha_i}, \qquad \alpha_{i+1} = d_{i+1} - \beta_i u_i. \end{aligned} \tag{9.6}$$

Domande:

a) Sia $A \in \mathbb{R}^{4\times 4}$ $A = \begin{pmatrix} 6 & -1 & 0 & 0 \\ 6 & -4 & -1 & 0 \\ 0 & -3 & -5 & 1 \\ 0 & 0 & -4 & 2 \end{pmatrix}$, e in particolare sia `d` il vettore contenente i termini sulla diagonale, sia `u` il vettore contenente i termini sulla sovra-diagonale e sia ℓ il vettore contenente i termini sulla sotto-diagonale.

b) Si determini il termine noto **b** in modo che la soluzione di riferimento del sistema $A\mathbf{x} = \mathbf{b}$ sia il vettore $\mathbf{xRif} = [1; 1; \ldots; 1] \in \mathbb{R}^4$

c) Si costruiscano la matrice L e la matrice U attraverso le relazioni in (9.6). Controllare che valga davvero $A = LU$.

d) Sfruttando la decomposizione $A = LU$ e le routine per la soluzione di sistemi triangolari inferiori e triangolari superiori risolvere il sistema $Ax = b$. (È quindi vietato l'uso del comando \ di MATLAB).

e) Tenendo conto della particolare struttura di L ed U (sono più che triangolari perché hanno molti zeri, in pratica hanno ciascuna solo 2 diagonali non nulle) risolvere i sistemi associati a queste matrici con un'opportuna strategia.

Appendice A

MATLAB: Introduzione (sintetica) al calcolo simbolico

Spesso è utile manipolare espressioni matematiche con l'ausilio del calcolatore per ottenere risultati in forma analitica. In questa sezione verranno mostrate alcune delle potenzialità del Symbolic Math Toolbox.

Il Symbolic Math Toolbox utilizza molti dei nomi delle funzioni numeriche di MATLAB e per ottenere le informazioni relative alla versione simbolica di una particolare funzione occorre digitare nella Command Window

```
help sym/nomefunzione
```

Il Symbolic Math Toolbox definisce un nuovo tipo di variabile, chiamato oggetto simbolico. É una struttura dati che memorizza una rappresentazione stringa del simbolo. Per creare oggetti simbolici in MATLAB si utilizzano i comandi `sym` e/o `syms`:

```
x = sym('x');
syms x;
```

Esistono vari comandi per manipolare espressioni simboliche E con variabile x

collect(E) raccoglie i coefficienti con la stessa potenza di x

expand(E) applica regole algebriche per espandere l'espressione E

factor(E) esprime E come prodotto di polinomi con coefficienti razionali

poly2sym(p) converte i coefficienti del vettore p in un polinomio simbolico

sym2poly(E) converte l'espressione E nel vettore di coefficienti

...

Data una funzione simbolica f essa può essere valutata in un punto tramite il comando `subs`, cioè si può assegnare un valore ad una variabile simbolica x:

```
subs(f,1)
subs(f,'x',1)
```

É inoltre possibile disegnare il grafico di questa funzione senza doverla valutare in un numero discreto di punti, usando il comando `ezplot` che disegna la funzione sull'intervallo standard $[-2\pi, 2\pi]$

```
ezplot(f)
```

altrimenti si può specificare l'intervallo

```
ezplot(f,[-5,5])
```

Dopo aver definito funzioni simboliche si possono effettuare diverse operazioni

diff(f) Restituisce la derivata dell'espressione f rispetto alla variabile indipendente di default (x) (si possono anche specificare altre variabili o l'ordine di derivata richiesta)

int(f) Restituisce l'integrale dell'espressione f (la sintassi può dipendere dalla versione di Matlab)

limit(f) Restituisce il valore del limite di f per x che tende a 0 (default)

...

A.1 Nota bene

In generale si preferisce tener separate parti di codice con l'intento di fare calcoli simbolici, dalle parti in cui si effettuano calcoli numerici standard.

In particolare bisogna fare attenzione al tipo di variabile considerato: se f ed x vengono dichiarate di tipo simbolico non possono essere utilizzate direttamente nei calcoli (cioè dopo o prima non possono essere vettori o funzioni, se non vengono in qualche modo 'puliti'). Su di esse si può operare solo con comandi simbolici, come quelli elencati sopra. Inoltre alcune funzioni hanno lo stesso nome (ma comportamenti differenti) se applicati a oggetti simbolici o ad oggetti standard: bisogna quindi fare attenzione ed essere consapevoli di quale è il proprio scopo.

A.2 Nota bene 2

E' consigliabile fare un uso moderato dei comandi simbolici. Spesso fare i calcoli a mano (se non sono troppo lunghi) permette di notare più facilmente alcune proprietà delle espressioni in oggetto.

Appendice B

MATLAB: le celle

Le celle sono contenitori, possiamo paragonarle a delle matrici (ma una cella non è necessariamente 2-dimensionale). Inoltre gli elementi di una cella non devono necessariamente essere dello stesso tipo e possono essere a loro volta oggetti complicati come matrici, vettori o funzioni.

- Per dichiarare una cella si può scrivere

```
A = cell(n,m)       % crea un contenitore n x m
A = cell(n,m,p)     % crea un contenitore n x m x p
A = cell(n,m,p,q)   % crea un contenitore n x m x p x q
```

- L'uso delle parentesi graffe o tonde per accedere ad *una* componente della cella, e modificarla, o per accedere contemporaneamente *a più* componenti della cella e modificarle dipende dalle situazioni. Vediamo degli esempi

```
A = cell(2,3);
% per mettere qualcosa in una singola posizione ci sono 2 modi equivalenti
A(2,3) = {5};
A{2,3} = 5;
A(1,2) = {[1,2,3]};
A{1,2} = [1,2,3];
A(2,2) = {@(x) 3*x + x^2};
A{2,2} = @(x) 3*x + x^2;

% per mettere lo stesso oggetto in piu' posizioni, solo 1 metodo
A(:,1) = {@(x) 3*x};

% per accedere ad 1 componente, 2 metodi
A{1,1}
A(1,1)

% ma se l'oggetto e' una funzione, l'unico modo per poter valutare una
% funzione in un punto e' con le graffe:
A{1,1}(2) % cioe' se nella posizione (1,1) ho  f = @(x) 3*x;
% scrivendo A{1,1}(2) eseguo ->  f(2)
```

Bibliografia

[1] S. De Marchi, "Appunti di Calcolo Numerico con codici in Matlab/Octave", 2009

[2] M. Caliari, "Dispense del corso di Laboratorio di Calcolo Numerico", 2008

[3] A. Quarteroni, "Introduzione al Calcolo Scientifico".

[4] A. Morzenti, A. Campi, E. Di Nitto, D. Loiacono, P. Spoletini "Introduzione alla programmazione in Malab", 2011 .

[5] M. Venturin, "Introduzione a Matlab", AA 2008 2009.

[6] V. Comincioli, “Analisi numerica: metodi, modelli, applicazioni”, 2005.