

Università degli studi di Firenze



Facoltà di Scienze Matematiche Fisiche e Naturali
Corso di laurea in Informatica

Relazione di Calcolo Numerico

A.A. 2015/2016

Autori:
Giulio Grimani
Marco Vignini

INDICE

Capitolo 1 – Errori ed aritmetica finita	2
Capitolo 2 – Radici di una equazione	14
Capitolo 3 – Sistemi lineari e nonlineari.....	25
Capitolo 4 – Approssimazione di funzioni	37
Capitolo 5 – Formule di quadratura	70
Capitolo 6 – Calcolo del Google pagerank	80
Codici Capitolo 2.....	96
Codici Capitolo 3.....	111
Codici Capitolo 4.....	118
Codici Capitolo 5.....	129
Codici Capitolo 6.....	132

ESERCIZI CAPITOLO 1

ERRORI ED ARITMETICA FINITA

Esercizio 1.1

Sia $x = e \approx 2.7182 = \tilde{x}$. Calcolare il corrispondente errore relativo \mathcal{E}_x . Verificare che il numero di cifre decimali corrette con cui \tilde{x} approssima x è all'incirca dato da:

$$-\log_{10}|\mathcal{E}_x|$$

Svolgimento

Ricordiamo la definizione di errore relativo:

$$\mathcal{E}_x = \frac{\tilde{x} - x}{x}$$

La rappresentazione del numero di Nepero e in MATLAB tramite la funzione $\exp(1)$ è: 2.718281828459046. Applicando la definizione abbiamo:

$$|\mathcal{E}_x| = \frac{|2.7182 - 2.718281828459046|}{|2.718281828459046|} = 3.010300778561018 * 10^{-5}$$

Il numero di cifre decimali corrette con cui \tilde{x} approssima x (nel nostro caso e) è 4, difatti:

$$-\log_{10}|\mathcal{E}_x| \approx 4.521$$

Per capire l'origine di tale approssimazione vale la pena ricordare che la precisione macchina u è una maggiorazione dell'errore relativo, e in caso di arrotondamento è data da:

$$u = \frac{1}{2}b^{1-m}$$

Pertanto in base $b = 2$ abbiamo:

$$2u = 2^{1-m} \Rightarrow 1 + \log_2 u = 1 - m \Rightarrow m = -\log_2 u$$

Esercizio 1.2

Dimostrare che se $f(x)$ è sufficientemente regolare e h è una quantità positiva "piccola", risulta

$$\frac{f(x_0 - 2h) - 4f(x_0 - h) + 3f(x_0)}{2h} = f'(x_0) + \mathcal{O}(h^2)$$

Svolgimento

La dimostrazione passa attraverso gli sviluppi di Taylor, che andiamo a definire.

Sia $f: [a, b] \rightarrow \mathbb{R}$ continua e derivabile n volte. Preso h tale che f sia definita in $[x_0 - h, x_0 + h]$, vale la formula:

$$f(x_0 + h) = \sum_{i=0}^{n-1} \frac{f^{(i)}(x_0)}{i!} h^i + R_n(h)$$

Dove R_n è il resto di ordine n , mentre x_0 è il centro dello sviluppo.

Pertanto:

$$\begin{aligned} f(x_0 - 2h) &= f(x_0) - 2f'(x_0)h + 2f''(x_0)h^2 + \mathcal{O}(h^3) \\ f(x_0 - h) &= f(x_0) - f'(x_0)h + \frac{1}{2}f''(x_0)h^2 + \mathcal{O}(h^3) \end{aligned}$$

Quindi:

$$\begin{aligned} &\frac{f(x_0) - 2f'(x_0)h + 2f''(x_0)h^2 - 4f(x_0) + 4f'(x_0)h - 2f''(x_0)h^2 + 3f(x_0) + \mathcal{O}(h^3)}{2h} = \\ &= \frac{2f'(x_0)h + \mathcal{O}(h^3)}{2h} = f'(x_0) + \mathcal{O}(h^2) \end{aligned}$$

Esercizio 1.3

Confrontare gli errori assoluti con cui le seguenti successioni approssimano $\sqrt{3}$ al quinto passo (usare $x_0 = 3$ per la prima successione e $x_0 = 3$ e $x_1 = 2$ per la seconda):

$$x_{k+1} = \frac{1}{2} \left(x_k + \frac{3}{x_k} \right) \qquad x_{k+1} = \frac{x_{k-1} x_k + 3}{x_{k-1} + x_k}$$

Svolgimento

Ricordiamo la definizione di errore assoluto:

$$\Delta x = \tilde{x} - x$$

Trattandosi di una successione, parliamo di errore assoluto di convergenza commesso ad ogni passo del metodo iterativo. Pertanto abbiamo:

$$\Delta x_k = x_k - x$$

Dove x_k è il risultato intermedio ed x il valore preciso d'approssimare, ovvero $\sqrt{3}$.

Per svolgere i conti è stato usato MATLAB. Presentiamo script e stampe relative alle due successioni:

```
clc;
f1 = @(x) (1/2)*(x + 3/x); % Prima successione
f2 = @(x0, x1) (x0*x1 + 3)/(x0 + x1); % Seconda successione
x = sqrt(3); % Valore esatto di x

xk = 3; % Innesco per la prima successione

disp('Prima successione:');
fprintf('k = 0\t\ttx(0) = 3\t\t\t\tdelta = |x(0) - x| = %.1d',abs(xk - x));
for k = 1 : 5
    xk = f1(xk);
    fprintf('\nk = %d\t\ttx(%d) = %.9f\t\tdelta = |x(%d) - x| = %.1d', k, k, xk,
        k, abs(xk - x));
end

x0 = 3; % Inneschi per la seconda successione
x1 = 2;

fprintf('\n\nSeconda successione:\n');
fprintf('k = 0\t\ttx(0) = 3\t\t\t\tdelta = |x(0) - x| = %.1d\n',abs(x0 - x));
fprintf('k = 1\t\ttx(1) = 2\t\t\t\tdelta = |x(1) - x| = %.1d',abs(x1 - x));
for k = 2 : 5
    xk = f2(x0, x1);
    fprintf('\nk = %d\t\ttx(%d) = %.9f\t\tdelta = |x(%d) - x| = %.1d', k, k, xk,
        k, abs(xk - x));
    x0 = x1;
    x1 = xk;
end
fprintf('\n');
```

```

Command Window

Prima successione:
k = 0      x(0) = 3      delta = |x(0) - x| = 1.3e+00
k = 1      x(1) = 2.000000000    delta = |x(1) - x| = 2.7e-01
k = 2      x(2) = 1.750000000    delta = |x(2) - x| = 1.8e-02
k = 3      x(3) = 1.732142857    delta = |x(3) - x| = 9.2e-05
k = 4      x(4) = 1.732050810    delta = |x(4) - x| = 2.4e-09
k = 5      x(5) = 1.732050808    delta = |x(5) - x| = 0

Seconda successione:
k = 0      x(0) = 3      delta = |x(0) - x| = 1.3e+00
k = 1      x(1) = 2      delta = |x(1) - x| = 2.7e-01
k = 2      x(2) = 1.800000000    delta = |x(2) - x| = 6.8e-02
k = 3      x(3) = 1.736842105    delta = |x(3) - x| = 4.8e-03
k = 4      x(4) = 1.732142857    delta = |x(4) - x| = 9.2e-05
k = 5      x(5) = 1.732050935    delta = |x(5) - x| = 1.3e-07
fx >> |

```

Confrontando gli errori delle due successioni si evince che la prima approssima meglio $\sqrt{3}$.

Esercizio 1.4

Dimostrare che, se un'aritmetica finita utilizza l'arrotondamento e una base b pari, detta u la precisione di macchina, risulta:

$$u = \frac{1}{2} b^{1-m}$$

dove m indica il numero di cifre (in base b) destinate alla rappresentazione della mantissa normalizzata del numero.

Svolgimento

Occorre qui ricordare molto brevemente come un computer rappresenta un numero reale $x \in I$, ovvero a come è definita la funzione $fl(x)$ che dall'insieme I degli infiniti numeri reali presenti nell'intervallo:

$$[-realMax, -realMin] \cup \{0\} \cup [realMin, realMax]$$

porta all'insieme finito \mathcal{M} dei numeri macchina. In particolare, ci interessa ricordare la politica adottata per la rappresentazione con arrotondamento.

Dato quindi un numero reale $x \in I$ scritto nella seguente forma normalizzata:

$$x = (\alpha_1.\alpha_2 \dots \alpha_m \alpha_{m+1} \dots) b^c \quad \text{con} \quad \alpha_1 \neq 0$$

definiamo la $fl(x)$ con rappresentazione per arrotondamento nel seguente modo:

$$fl(x) = \alpha_1.\alpha_2 \dots \tilde{\alpha}_m$$

dove:

$$\tilde{\alpha}_m = \begin{cases} \alpha_m & \text{se } \alpha_{m+1} < \frac{b}{2} \\ \alpha_m + 1 & \text{se } \alpha_{m+1} \geq \frac{b}{2} \end{cases}$$

Ricordandoci inoltre che:

$$fl(x) = x(1 + \epsilon_x) \quad \text{ovvero} \quad |\epsilon_x| = \frac{|x - fl(x)|}{|x|}$$

siamo ora in grado di dimostrare quanto richiesto.

Primo caso: $\alpha_{m+1} < \frac{b}{2} \Rightarrow \tilde{\alpha}_m = \alpha_m$

$$|\epsilon_x| = \frac{(\alpha_1.\alpha_2 \dots \alpha_m \alpha_{m+1} \dots - \alpha_1.\alpha_2 \dots \alpha_m) b^c}{(\alpha_1.\alpha_2 \dots \alpha_m \alpha_{m+1} \dots) b^c} = \frac{\overbrace{(0.0 \dots 0 \alpha_{m+1} \alpha_{m+2} \dots)}^{m \text{ zeri}}}{(\alpha_1.\alpha_2 \dots \alpha_m \alpha_{m+1} \dots)} =$$

$$= \frac{(\alpha_{m+1} \cdot \alpha_{m+2} \dots) b^{-m}}{(\alpha_1 \cdot \alpha_2 \dots \alpha_m \alpha_{m+1} \dots)}$$

Quel che vogliamo fare ora è ottenere una maggiorazione di tale quantità. Osserviamo che:

$$1 \leq \alpha_i < b$$

ed in particolare:

$$\alpha_{m+1} < \frac{b}{2}$$

pertanto possiamo maggiorare il termine a sinistra ponendo uguale a 1 il denominatore e uguale a $\frac{1}{2}b$ il numeratore:

$$\frac{(\alpha_{m+1} \cdot \alpha_{m+2} \dots) b^{-m}}{(\alpha_1 \cdot \alpha_2 \dots \alpha_m \alpha_{m+1} \dots)} \leq \frac{1}{2} b * b^{-m} = \frac{1}{2} b^{1-m}$$

Chiameremo tale risultato *precisione di macchina* u . Notiamo quindi che $|\mathcal{E}_x| \leq u$.

Secondo caso: $\alpha_{m+1} \geq \frac{b}{2} \Rightarrow \tilde{\alpha}_m = \alpha_m + 1$

Il primo passaggio è analogo, ma si perviene a:

$$\frac{(0.0 \dots 0 \overset{m}{\underset{\sim}{1}} - 0.0 \dots \overset{m}{\underset{\sim}{0}} \alpha_{m+1} \dots)}{(\alpha_1 \cdot \alpha_2 \dots \alpha_m \alpha_{m+1} \dots)} = \frac{(1 - 0. \alpha_{m+1} \dots) b^{1-m}}{(\alpha_1 \cdot \alpha_2 \dots \alpha_m \alpha_{m+1} \dots)}$$

Qui ci basta osservare che $\alpha_{m+1} \geq \frac{b}{2}$, pertanto la quantità $(1 - 0. \alpha_{m+1} \dots)$ sarà almeno minore o uguale di $\frac{1}{2}$. Analogamente a prima, si ha che:

$$\frac{(1 - 0. \alpha_{m+1} \dots) b^{1-m}}{(\alpha_1 \cdot \alpha_2 \dots \alpha_m \alpha_{m+1} \dots)} \leq \frac{1}{2} b^{1-m}$$

Esercizio 1.5

Calcolare il numero di cifre binarie utilizzate per rappresentare la mantissa di un numero, sapendo che la precisione di macchina è $u \approx 3.05 * 10^{-5}$ e che si usa l'arrotondamento.

Svolgimento

Usando l'arrotondamento, con $b = 2$, sappiamo che:

$$u = \frac{1}{2} 2^{1-m}$$

pertanto:

$$\frac{1}{2} 2^{1-m} \approx 3.05 * 10^{-5}$$

$$2^{-m} \approx 3.05 * 10^{-5}$$

Passando ai logaritmi abbiamo:

$$-m \approx \log_2(3.05 * 10^{-5})$$

$$m \approx -\log_2(3.05 * 10^{-5}) \approx 15.00083$$

Pertanto, con 16 bit di mantissa, si è sicuri di poter rappresentare ogni numero con la precisione macchina data.

Esercizio 1.6

Con riferimento allo standard *IEEE 754* determinare, relativamente alla singola precisione, il più grande numero di macchina e la precisione di macchina.

Svolgimento

Ricordiamo le specifiche che ci interessano dello standard *IEEE 754* relative alla singola precisione:

- Base $b = 2$
- Rappresentazione per arrotondamento
- 24 bit per la mantissa m più 8 bit per la caratteristica s , per un totale di 32 bit
- *Shift* v pari a 127 se la mantissa è normalizzata

Ricordiamo ovviamente la formula per il più grande numero di macchina *realMax*:

$$realMax = (1 - b^{-m}) * b^{\varphi} \quad \text{con} \quad \varphi = b^s - v$$

Ricordiamo infine che la precisione di macchina u , nel caso di rappresentazione per arrotondamento, è la seguente:

$$u = \frac{1}{2} b^{1-m}$$

Pertanto abbiamo che:

$$realMax = (1 - 2^{-24}) * 2^{129} \approx 6.8 * 10^{38}$$

$$u = \frac{1}{2} * 2^{-23} = 2^{-24} \approx 5.9 * 10^{-8}$$

Esercizio 1.7

Eseguire le seguenti istruzioni MATLAB:

```
x = 0;  
delta = 1/7;  
while x ~= 1  
    x = x + delta  
end
```

Spiegarne il non funzionamento.

Svolgimento

Aniché uscire alla settima iterazione come ci si potrebbe aspettare, il codice va in loop. Questo è dovuto alla rappresentazione in macchina del numero $\frac{1}{7}$, che essendo periodico in base 10 lo è anche in base 2_{c2} . L'errore generato dalla rappresentazione di tali numeri sul calcolatore rende estremamente poco affidabili controlli basati sulla diversità (o uguaglianza) di valori, specialmente se interi. Per ovviare a questo problema si può pensare di modificare tale controllo introducendo una tolleranza presso la quale sia ragionevole accontentarsi dell'approssimazione ottenuta: $abs(x - 1) > eps$

Esercizio 1.8

Eseguire le seguenti istruzioni MATLAB:

```
ep = 1;  
while ep + 1 > 1  
    ep = ep/2;  
end  
ep = ep*2;
```

Confrontare il risultato che si ottiene con il valore della costante predefinita *eps* e dedurre il significato di *ep*.

Svolgimento

L'esecuzione del codice porta ad avere $ep = eps$.

La condizione d'uscita del ciclo ci dice sostanzialmente che si esce quando *ep* diventa così piccolo che la sua somma con 1 non viene più avvertita dal calcolatore come qualcosa maggiore di 1.

All'interno del ciclo *ep* viene ogni volta dimezzato. Questo significa fare le divisioni ripetute per 2. Difatti non è un caso che si esca dal ciclo giunti alla 53-esima iterazione: 53 è il numero di bit dedicati alla mantissa nello standard *IEEE 754* a doppia precisione.

La moltiplicazione per 2 fuori dal ciclo compensa il fattore $\frac{1}{2}$ che troviamo nella definizione di precisione macchina con rappresentazione per arrotondamento ($u = \frac{1}{2} b^{1-m}$), ovvero quella usata dallo standard.

Esercizio 1.9

Scrivere l'algoritmo più efficace per evitare problemi intermedi di overflow per calcolare, in aritmetica finita, l'espressione $\sqrt{x^4 + y^4}$.

Svolgimento

Come evidenziato dalla traccia, l'espressione potrebbe essere soggetta ad overflow nel caso in cui uno o entrambi tra x e y siano maggiori del più alto numero rappresentabile in macchina $realMax$. Come già discusso nell'esercizio 4, un numero n è rappresentabile in macchina sse:

$$n \in I = [-realMax, -realMin] \cup \{0\} \cup [realMin, realMax]$$

Definendo quindi $M = \max\{|x|, |y|\}$ possiamo manipolare l'espressione come segue:

$$\sqrt{x^4 + y^4} = \sqrt{M^4 \left(\frac{x^4}{M^4} + \frac{y^4}{M^4} \right)} = |M^2| \sqrt{\left| \frac{x}{M} \right|^4 + \left| \frac{y}{M} \right|^4}$$

In questo modo abbiamo ridotto il termine maggiore di due gradi, e dentro la radice un operando sarà uguale a 1 mentre l'altro sarà l'elevamento a potenza quarta di qualcosa compreso tra 0 e 1.

Esercizio 1.10

Eeguire l'analisi dell'errore relativo nel calcolo della seguente espressione:

$$(x \oplus y) \oplus z$$

Dedurre che la somma non gode in macchina della proprietà associativa e che, se si devono sommare tre numeri concordi, conviene sommare prima quelli più piccoli in valore assoluto.

Svolgimento

Per convincersi del fatto che la somma in macchina non gode della proprietà associativa, ovvero che:

$$(x \oplus y) \oplus z \neq x \oplus (y \oplus z)$$

ci basta trascrivere i termini in floating point:

$$(x \oplus y) \oplus z = fl(fl(fl(x) + fl(y)) + fl(z)) =$$

$$= \left((x(1 + \varepsilon_x) + y(1 + \varepsilon_y))(1 + \varepsilon_1) + z(1 + \varepsilon_z) \right) (1 + \varepsilon_2)$$

mentre:

$$\begin{aligned} x \oplus (y \oplus z) &= fl(fl(x) + fl(fl(y) + fl(z))) = \\ &= \left(x(1 + \varepsilon_x) + (y(1 + \varepsilon_y) + z(1 + \varepsilon_z))(1 + \varepsilon_1) \right) (1 + \varepsilon_2) \end{aligned}$$

Tutto ciò risulta ancor più evidente svolgendo l'analisi dell'errore relativo. Prendiamo ad esempio:

$$R = x + y + z$$

e facciamo l'analisi di:

$$\tilde{R} = (x \oplus y) \oplus z$$

$$\begin{aligned} \tilde{R} &= \left((x + x\varepsilon_x + y + y\varepsilon_y)(1 + \varepsilon_1) + z + z\varepsilon_z \right) (1 + \varepsilon_2) \quad \begin{array}{l} \text{trascuriamo} \\ \text{i termini con} \\ \varepsilon \text{ quadratico} \end{array} \approx \\ &\approx (x + x\varepsilon_x + y + y\varepsilon_y + x\varepsilon_1 + y\varepsilon_1 + z + z\varepsilon_z)(1 + \varepsilon_2) \approx \\ &\approx x + y + z + x\varepsilon_x + y\varepsilon_y + z\varepsilon_z + x\varepsilon_1 + y\varepsilon_1 + x\varepsilon_2 + y\varepsilon_2 + z\varepsilon_2 = \dots \end{aligned}$$

Notiamo che $x + y + z = R$. Raccogliamo inoltre ε_1 ed ε_2 .

$$\dots = R + x\varepsilon_x + y\varepsilon_y + z\varepsilon_z + (x + y)\varepsilon_1 + (x + y + z)\varepsilon_2 \approx \tilde{R}$$

Passiamo all'analisi dell'errore relativo:

$$\left| \frac{\tilde{R} - R}{R} \right| \approx \frac{|x\varepsilon_x + y\varepsilon_y + z\varepsilon_z + (x + y)\varepsilon_1 + (x + y + z)\varepsilon_2|}{|x + y + z|} \leq$$

Maggioriamo il termine con la disuguaglianza triangolare, raccogliendo al contempo un

$\varepsilon_{max} = \max\{|\varepsilon_x|, |\varepsilon_y|, |\varepsilon_z|, |\varepsilon_1|, |\varepsilon_2|\}$. Pertanto abbiamo:

$$\begin{aligned} &\leq \frac{|x| + |y| + |z| + |x + y| + |x + y + z|}{|x + y + z|} \varepsilon_{max} = \\ &= \left(\frac{|x| + |y| + |z|}{|x + y + z|} + \frac{|x + y|}{|x + y + z|} + 1 \right) \varepsilon_{max} \geq \left| \frac{\tilde{R} - R}{R} \right| \end{aligned}$$

Svolgendo l'analisi di:

$$x \oplus (y \oplus z)$$

otteniamo invece questo risultato:

$$\dots = \left(\frac{|x| + |y| + |z|}{|x + y + z|} + \frac{|y + z|}{|x + y + z|} + 1 \right) \varepsilon_{max} \geq \left| \frac{\tilde{R} - R}{R} \right|$$

Si evince facilmente come i termini in grassetto dipendano da come viene impostata la somma.

Dedurre che convenga sommare prima i termini più piccoli risulta più immediato facendo un esempio con numeri concreti. Generalmente parlando, quel che si cerca di fare è sommare prima i due numeri piccoli in modo tale da aumentare le possibilità del calcolatore di “percepire” tale quantità nella somma col numero grande. Si immagini un numero grande con una determinata caratteristica, e si immagini di sommare a tale numero una quantità così piccola (quindi con una caratteristica di diversi ordini di grandezza inferiore) da non alterare neanche l’ultima cifra rappresentata dopo la virgola nella rappresentazione di tale somma, che sarà un numero normalizzato, quindi moltiplicato per la base elevata alla caratteristica maggiore. In tal caso, il risultato sarebbe il numero grande stesso, e la somma avrebbe perso di significato.

Ad esempio, si supponga di avere un calcolatore che operi in base 10, con 4 cifre di mantissa e politica di arrotondamento.

$$x + y + z$$

$$\begin{aligned} x &= 2.7351 * 10^2 & fl(x) &= 2.735 * 10^2 \\ y &= 4.8016 * 10^{-2} & fl(y) &= 4.802 * 10^{-2} \\ z &= 3.6154 * 10^{-2} & fl(z) &= 3.615 * 10^{-2} \end{aligned}$$

$$(x \oplus y) \oplus z$$

$$x \oplus y = fl(2.735 * 10^2 + 4.802 * 10^{-2}) = fl(2.7354802 * 10^2) = 2.735 * 10^2$$

Tale risultato corrisponde proprio alla $fl(x)$. Già da questo risultato intermedio ci accorgiamo che non è stata percepita la somma. Stessa sorte subirà la successiva somma con la $fl(z)$, e si avrà che $(x \oplus y) \oplus z = fl(x)$.

Se invece facciamo:

$$x \oplus (y \oplus z)$$

$$y \oplus z = fl(4.802 * 10^{-2} + 3.615 * 10^{-2}) = fl(8.417 * 10^{-2}) = 8.417 * 10^{-2}$$

Sommando tale quantità alla $fl(x)$, siamo riusciti a far “percepire” la somma al calcolatore:

$$fl(2.735 * 10^2 + 8.417 * 10^{-2}) = fl(2.7358417 * 10^2) = 2.736 * 10^2 \neq fl(x)$$

Esercizio 1.11

Eseguire le seguenti istruzioni MATLAB:

```
format long e  
a = 0.3  
b = 0.29999999  
y = a - b
```

Dire se a e b sono numeri di macchina (tenendo presente che si lavora in base 2) e stimare l'errore relativo commesso dal calcolatore nella loro memorizzazione. Valutare quindi l'errore relativo sul risultato (caso di cancellazione numerica) e verificare che è in accordo con l'analisi del condizionamento effettuata.

Svolgimento

Non sono numeri perfettamente rappresentabili in macchina in quanto le loro corrispondenti frazioni presentano al denominatore un numero non divisibile per una potenza di 2. Si nota inoltre che a in base 2 è periodico:

n	$n * 2$	$trunc$
0.3	0.6	0
0.6	1.2	1
0.2	0.4	0
0.4	0.8	0
0.8	1.6	1
0.6		

Pertanto una maggiorazione dell'errore commesso dal calcolatore nella loro memorizzazione è la precisione macchina u .

Per quanto riguarda l'analisi del condizionamento della somma generica:

$$y = a + b$$

quel che vogliamo fare è mettere in relazione l'errore relativo sui dati in input con quello sui dati in output. Passando quindi all'aritmetica finita dei calcolatori abbiamo:

$$\tilde{y} = \tilde{a} + \tilde{b}$$

dove:

$$\tilde{y} = y(1 + \epsilon_y)$$

$$\tilde{a} = a(1 + \epsilon_a)$$

$$\tilde{b} = b(1 + \epsilon_b)$$

Avremo quindi:

$$y(1 + \varepsilon_y) = a(1 + \varepsilon_a) + b(1 + \varepsilon_b) = \overbrace{a + b}^{=y} + a\varepsilon_a + b\varepsilon_b$$
$$y + y\varepsilon_y = y + a\varepsilon_a + b\varepsilon_b$$
$$\varepsilon_y = \frac{a\varepsilon_a + b\varepsilon_b}{a + b}$$

Maggioriamo usando la disuguaglianza triangolare, raccogliendo al contempo un errore relativo $\varepsilon_{max} = \max\{|\varepsilon_a|, |\varepsilon_b|\}$

$$|\varepsilon_y| \leq \frac{|a| + |b|}{|a + b|} \varepsilon_{max} \equiv k \varepsilon_{max}$$

Chiamiamo k il *fattore di condizionamento* del problema in esame.

Si evince subito che, in caso di addendi di segno concorde, il problema è ben condizionato in quanto $k = 1$. Viceversa, è mal condizionato quando $a \approx -b$, ovvero quando $k \rightarrow \infty$: l'errore sui dati in output viene amplificato da un fattore arbitrariamente elevato rispetto all'errore sui dati in input. Questo dà origine al fenomeno della cancellazione numerica. Nel nostro caso specifico abbiamo:

$$|\varepsilon_y| \leq \frac{|0.3| + |0.2999999|}{|0.3 - 0.2999999|} u = \frac{0.5999999}{0.0000001} u = 5999999 * u$$

Risulta ora facile verificare come l'errore relativo ε_y sul risultato sia in accordo con l'analisi del condizionamento del problema in esame:

$$y = a - b = 0.3 - 0.2999999 = 0.0000001 = 10^{-7}$$

Mostriamo quindi sulla finestra di comando MATLAB l'esecuzione delle seguenti istruzioni:

```
Command Window
>> yTilde = a - b;
>> errRelY = (yTilde - 10^-7)/10^-7

errRelY =

    2.875570976809943e-11

>> 5999999 * eps

ans =

    1.332267407505583e-09

fx >> |
```

Notiamo come $k * u$ sia di due ordini di grandezza più grande di ε_y .

ESERCIZI CAPITOLO 2

RADICI DI UNA EQUAZIONE

Esercizio 2.1

Definire una procedura iterativa basata sul metodo di Newton per determinare $\sqrt[n]{\alpha}$, con $\alpha > 0$, implementarla in MATLAB e valutarne le prestazioni per approssimare $\sqrt[3]{10}$.

Svolgimento

Il problema preso in esame in questo capitolo affronta la risoluzione dell'equazione:

$$f(x) = 0, \quad x \in \mathbb{R}, \quad f: \mathbb{R} \rightarrow \mathbb{R}$$

Siamo quindi interessati alla determinazione della radice di tale equazione attraverso l'utilizzo di procedure iterative che, generalmente parlando, dato un punto d'innesco calcolano ad ogni iterazione un'approssimazione auspicabilmente sempre più accurata della radice. La procedura richiesta dalla traccia è basata sul metodo di Newton, pertanto ne introduciamo l'espressione:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \quad i = 0, 1, 2 \dots$$

Osserviamo subito che l'espressione è definita solo se $f'(x_i) \neq 0$.

Dal momento che vogliamo determinare $\sqrt[n]{\alpha}$ come radice, il primo passo è quello di trovare una f t.c. $f(\sqrt[n]{\alpha}) = 0$, e la individuiamo in $f(x) = x^n - \alpha$. Quindi $f'(x) = nx^{n-1}$, e dovendo essere $f'(x_i) \neq 0$ avremo $x \neq 0$ e $n \neq 0$.

Possiamo ora comporre l'espressione, manipolandola opportunamente per evitare cancellazione numerica e ridurre il più possibile il numero di operazioni da fare ad ogni iterazione:

$$\begin{aligned} x_{i+1} &= x_i - \frac{x_i^n - \alpha}{nx_i^{n-1}} = \frac{nx_i^n - x_i^n + \alpha}{nx_i^{n-1}} = \frac{1}{n} \left(\frac{(n-1)x_i^n + \alpha}{x_i^{n-1}} \right) = \frac{1}{n} \left(\frac{(n-1)x_i^n}{x_i^{n-1}} + \frac{\alpha}{x_i^{n-1}} \right) = \\ &= \frac{1}{n} \left((n-1)x_i + \frac{\alpha}{x_i^{n-1}} \right) \end{aligned}$$

Siamo finalmente pronti ad implementare la relativa funzione MATLAB per valutarne le prestazioni nell'approssimazione di $\sqrt[3]{10} \approx 2.1544$. Questo significa $n = 3$ e $\alpha = 10$. La funzione diventa cubica e, data la natura di tali funzioni, discuteremo alcuni punti di innesco significativi: $x_0 = -10, -3, -1.71, -1, -1, 10^{-7}, 1, 2, \sqrt[3]{10}, 3, 10, 50, 100, 50000$.

Il criterio d'arresto scelto è quello dell'incremento relativo, ovvero con l'uso di una $rtolx$ così come indicato nella formula 2.15 di pagina 35 del libro di testo, con $tolx$ iniziale pari a 10^{-10} .

Il grafico della funzione:

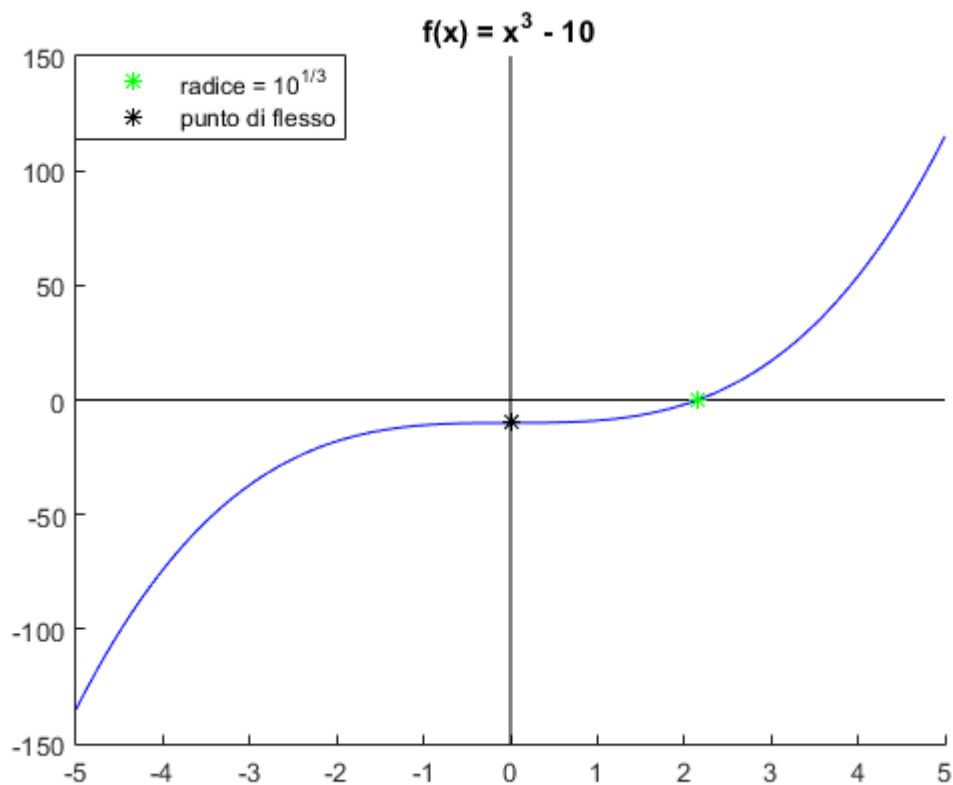


Tabella delle prestazioni:

Punto d'innescio x_0	Numero di iterazioni
-10	13
-3	18
-1.71	57
-1	6
10^{-7}	87
1	8
2	5
$\sqrt[3]{10}$	1
3	6
10	9
50	13
100	15
50000	30

Notiamo innanzitutto che per $x_0 \rightarrow 0$, ovvero in prossimità del punto di flesso, le iterazioni aumentano così come ci si aspetterebbe, poiché in quel punto la tangente diventa orizzontale. Contrariamente a quanto ci direbbe l'intuito, per $x_0 < 0$ il metodo non impiega meno iterazioni se l'innescio è più vicino alla radice, bensì si tratta di vedere come “rimbalza” la tangente, pertanto la convergenza non è assicurata. Per $x_0 \in (0, \sqrt[3]{10}]$ il numero di iterazioni diventa, coerentemente con quanto appena osservato, sempre più piccolo man mano che ci allontaniamo dallo 0, per poi cominciare a divergere (lentamente) per $x_0 > \sqrt[3]{10}$.

L'implementazione della funzione in MATLAB è provvista di parte grafica e varie stampe di output sulla finestra di comando, tra cui il numero di iterazioni, le x_i intermedie ed il tempo d'esecuzione. Il codice è posto nell'ultima sezione della relazione, ed è immediatamente raggiungibile cliccando su questo riferimento: [Codice MATLAB 2.1](#).

Esercizio 2.2

Definire una procedura iterativa basata sul metodo delle secanti per determinare $\sqrt[n]{\alpha}$, con $\alpha > 0$, implementarla in MATLAB e confrontare le prestazioni con quelle della procedura costruita al punto precedente per approssimare $\sqrt[3]{10}$ (usare il medesimo criterio di arresto).

Svolgimento

Enunciamo la formula del metodo iterativo delle secanti:

$$x_{i+1} = \frac{f(x_i)x_{i-1} - f(x_{i-1})x_i}{f(x_i) - f(x_{i-1})}, \quad i = 0, 1, 2, \dots$$

Notiamo innanzitutto che, a differenza del metodo di Newton, questo metodo lavora su due approssimazioni della radice. Analogamente all'esercizio precedente, ricordandoci che $f(x) = x^n - \alpha$, manipoliamo anche qui l'espressione per giungere ad una forma più conveniente:

$$\begin{aligned} x_{i+1} &= \frac{(x_i^n - \alpha)x_{i-1} - (x_{i-1}^n - \alpha)x_i}{x_i^n - \alpha - x_{i-1}^n + \alpha} = \frac{x_{i-1}x_i^n - \alpha x_{i-1} - x_{i-1}^n x_i + \alpha x_i}{x_i^n - x_{i-1}^n} \\ &= \frac{(x_i^{n-1} - x_{i-1}^{n-1})x_{i-1}x_i + \alpha(x_i - x_{i-1})}{x_i^n - x_{i-1}^n} = \dots \end{aligned}$$

La differenza di due potenze con lo stesso esponente positivo è sempre divisibile per la differenza delle basi, e $\forall n > 0$ si ha che:

$$a^n - b^n = (a - b)(a^{n-1} + a^{n-2}b + a^{n-3}b^2 + \dots + a^2b^{n-3} + ab^{n-2} + b^{n-1})$$

quindi la nostra espressione diventa:

$$\begin{aligned} \dots &= \frac{(x_i - x_{i-1})(x_i^{n-2} + x_{i-1}x_i^{n-3} + \dots + x_{i-1}^{n-3}x_i + x_{i-1}^{n-2})x_{i-1}x_i + \alpha(x_i - x_{i-1})}{(x_i - x_{i-1})(x_i^{n-1} + x_{i-1}x_i^{n-2} + \dots + x_{i-1}^{n-2}x_i + x_{i-1}^{n-1})} = \\ &= \frac{(x_i - x_{i-1}) \left((x_i^{n-2} + x_{i-1}x_i^{n-3} + \dots + x_{i-1}^{n-3}x_i + x_{i-1}^{n-2})x_{i-1}x_i + \alpha \right)}{(x_i - x_{i-1})(x_i^{n-1} + x_{i-1}x_i^{n-2} + \dots + x_{i-1}^{n-2}x_i + x_{i-1}^{n-1})} \\ &= \frac{(x_i^{n-2} + x_{i-1}x_i^{n-3} + \dots + x_{i-1}^{n-3}x_i + x_{i-1}^{n-2})x_{i-1}x_i + \alpha}{x_i^{n-1} + x_{i-1}x_i^{n-2} + \dots + x_{i-1}^{n-2}x_i + x_{i-1}^{n-1}} \end{aligned}$$

Siamo finalmente pronti a implementare il metodo: [Codice MATLAB 2.2](#). Come suggerito dal libro di testo, per trovare la seconda approssimazione è stato utilizzato un passo del metodo di Newton. Per la scomposizione della differenza di due potenze è stato appositamente creato

un mini metodo: [Codice MATLAB 2.2.1](#). Stampe, grafica e criterio d'arresto sono analoghi all'esercizio precedente.

Per dare un senso al confronto, riportiamo nella tabella solo le prestazioni per quei valori in cui la funzione è sufficientemente regolare.

Tabella delle prestazioni:

Punto d'innescio x_0	Iterazioni - Newton	Iterazioni - Secanti
$\sqrt[3]{10}$	1	1
3	6	6
10	9	11
50	13	17
100	15	19
50000	30	41
100000	32	44
1000000	37	52

Così come ci aspettavamo, più ci allontaniamo dalla radice più risulta evidente dal numero di iterazioni che il metodo di Newton converge più rapidamente di quello delle secanti. Difatti, se $f(x)$ è sufficientemente regolare, il metodo di Newton converge quadraticamente verso radici semplici, mentre quello delle secanti ha un ordine di convergenza pari solo a $\frac{\sqrt{5}+1}{2} \approx 1.618$.

Esercizio 2.3

Implementare in MATLAB (*functions*) i metodi di Newton, di Newton modificato e di accelerazione di Aitken combinato a Newton. Come criterio di arresto usare il criterio dell'incremento. Per Newton prevedere anche la possibilità di usare il criterio modificato come specificato al termine della Sezione 2.6 del libro (stimando la costante asintotica c).

Svolgimento

Per quanto riguarda i metodi di Newton abbiamo pensato di fare un'unica *function* che, a seconda degli input dell'utente, svolgesse quanto richiesto. La costante asintotica, qualora richiesta, verrà stampata sulla finestra di comando insieme agli altri soliti output.

[Codice MATLAB 2.3.1](#) – Newton

[Codice MATLAB 2.3.2](#) – Aitken

Entrambi i codici utilizzano il criterio d'arresto usato negli esercizi precedenti, ed hanno una parte grafica. Sono entrambi provvisti di controlli sulla derivata prima e, nel caso di Aitken, viene controllato il denominatore della funzione d'iterazione per evitare la cancellazione numerica.

Esercizio 2.4

Partendo da $x_0 = 20$, confrontare le prestazioni dei metodi di Newton, di Newton modificato e di accelerazione di Aitken per approssimare lo zero multiplo della seguente funzione:

$$f(x) = (x - 1)^{20} \exp(x)$$

Usare il criterio dell'incremento per tutti e tre i metodi. Quindi per Newton usare anche il criterio modificato e stabilire se e quali differenze comporta sulle prestazioni del metodo (anche in termini di accuratezza finale dell'approssimazione ottenuta). Verificare che il valore stimato di c converge a $\frac{19}{20}$ (perché?).

Svolgimento

Si nota facilmente che la molteplicità della radice è 20, e la radice è 1. Per il confronto è stato usato il solito criterio d'arresto dell'incremento basato sull'introduzione di una $rtolx$ su un massimo di 1000 iterazioni. Riportiamo quanto ottenuto in una tabella, dove:

m = molteplicità della radice, c = costante asintotica, i = # iterazioni ed n = # di valutazioni di funzione.

Per $x_0 = 20$ abbiamo:

<i>tolx</i>	<i>Aitken</i>	<i>Newton m</i>	<i>Newton</i>	<i>Newton c</i>
10^{-1}	$\tilde{x} \approx 0.99$ $i = 6 \quad n = 24$	$\tilde{x} \approx 1 + 2 * 10^{-6}$ $i = 5 \quad n = 10$	$\tilde{x} \approx 19.5$ $i = 1 \quad n = 2$	$\tilde{x} \approx 19.5$ $i = 1 \quad n = 2 \quad c \approx \dots$
10^{-2}	$\tilde{x} \approx 1$ $i = 7 \quad n = 28$	$\tilde{x} \approx 1 + 2 * 10^{-6}$ $i = 5 \quad n = 10$	$\tilde{x} \approx 1.4$ $i = 91 \quad n = 182$	$\tilde{x} \approx 1 + 2 * 10^{-2}$ $i = 153 \quad n = 306 \quad c \approx 0.95 + 1 * 10^{-4}$
10^{-3}	\vdots	$\tilde{x} \approx 1 + 2 * 10^{-13}$ $i = 6 \quad n = 12$	$\tilde{x} \approx 1 + 3 * 10^{-2}$ $i = 141 \quad n = 282$	$\tilde{x} \approx 1 + 2 * 10^{-3}$ $i = 198 \quad n = 396 \quad c \approx 0.95 + 1 * 10^{-5}$
10^{-4}	\vdots	$\tilde{x} \approx 1 + 2 * 10^{-13}$ $i = 6 \quad n = 12$	$\tilde{x} \approx 1 + 3 * 10^{-3}$ $i = 186 \quad n = 372$	$\tilde{x} \approx 1 + 2 * 10^{-4}$ $i = 243 \quad n = 486 \quad c \approx 0.95 + 1 * 10^{-6}$
10^{-5}	\vdots	$\tilde{x} \approx 1 + 2 * 10^{-13}$ $i = 6 \quad n = 12$	$\tilde{x} \approx 1 + 3 * 10^{-4}$ $i = 231 \quad n = 462$	$\tilde{x} \approx 1 + 2 * 10^{-5}$ $i = 288 \quad n = 576 \quad c \approx 0.95 + 1 * 10^{-7}$
10^{-6}	$\tilde{x} = x^* = 1$ $i = 8 \quad n = 30$	$\tilde{x} = x^* = 1$ $i = 7 \quad n = 14$	$\tilde{x} \approx 1 + 3 * 10^{-5}$ $i = 276 \quad n = 552$	$\tilde{x} \approx 1 + 2 * 10^{-6}$ $i = 333 \quad n = 666 \quad c \approx 0.95 + 1 * 10^{-8}$
10^{-7}	\vdots	\vdots	$\tilde{x} \approx 1 + 3 * 10^{-6}$ $i = 321 \quad n = 642$	$\tilde{x} \approx 1 + 2 * 10^{-7}$ $i = 378 \quad n = 756 \quad c \approx 0.94999998$
10^{-8}	\vdots	\vdots	$\tilde{x} \approx 1 + 3 * 10^{-7}$ $i = 365 \quad n = 730$	$\tilde{x} \approx 1 + 2 * 10^{-8}$ $i = 423 \quad n = 846 \quad c \approx 0.94999996$
10^{-9}	\vdots	\vdots	$\tilde{x} \approx 1 + 3 * 10^{-8}$ $i = 410 \quad n = 820$	$\tilde{x} \approx 1 + 2 * 10^{-9}$ $i = 468 \quad n = 936 \quad c \approx 0.949998$
10^{-10}	\vdots	\vdots	$\tilde{x} \approx 1 + 3 * 10^{-9}$ $i = 455 \quad n = 910$	$\tilde{x} \approx 1 + 2 * 10^{-10}$ $i = 513 \quad n = 1026 \quad c \approx 0.94998$
<i>eps</i>	\vdots	\vdots	$\tilde{x} \approx 1 + 1 * 10^{-14}$ $i = 704 \quad n = 1408$	$\tilde{x} \approx 1 + 6 * 10^{-15}$ $i = 714 \quad n = 1428 \quad c = 0.5$

Ci sono diverse considerazioni da fare. Partiamo dalle più ovvie: i metodi a convergenza quadratica (Aitken e Newton modificato) sono chiaramente più accurati e veloci di quelli a convergenza lineare. Al costo di un maggior numero di valutazioni di funzioni, Aitken è più preciso, tant'è che con $tolx = 10^{-6}$ si esce al controllo $fx == 0$ ad inizio ciclo, che spiega l'ulteriore iterazione. Coerentemente con quanto dice la teoria, Newton lineare con costante asintotica risulta essere più preciso di Newton lineare, al costo ovviamente di qualche iterata (e quindi anche di qualche valutazione di funzione) in più.

È interessante soffermarci un attimo sui risultati ottenuti con $tolx = 10^{-1}$: questi son dovuti alla scelta del criterio d'arresto (2.15) a pag. 35 del libro di testo, ovvero:

$$\frac{|e_i|}{tolx + rtolx|x^*|} \rightarrow \frac{|x_{i+1} - x_i|}{1 + |x_{i+1}|} \leq tolx$$

dove *'le approssimazioni $x^* \approx x_{i+1}$ e $|e_i| \approx |x_{i+1} - x_i|$ sono convenientemente utilizzabili. Inoltre la scelta $rtolx = tolx$ è spesso considerata'*. Abbiamo verificato sperimentalmente che, in effetti, usando il criterio d'arresto dell'incremento "assoluto" $|x_{i+1} - x_i| \leq tolx$, i risultati sono estremamente di poco più precisi, ma al costo di ben 14 iterate in più (e quindi relative valutazioni di funzione). Quel che però qui accade è che, per come è fatta la funzione, l'assunzione $x^* \approx x_{i+1}$ risulta essere troppo superficiale: la tangente in $x_0 = 20$ interseca l'asse delle x a circa 19.5, ed il denominatore derivante riesce a ridurre la frazione a qualcosa di minore di 10^{-1} , facendo terminare la procedura iterativa e "falsando" quindi il risultato. Per questa funzione, e con questo punto d'innesco, ciò accade solo per una $tolx = 10^{-1}$. Da questo si deduce che la scelta del criterio d'arresto ottimale non può prescindere dallo studio del particolare problema.

Concludiamo osservando che, per le caratteristiche della funzione data, vale qui il teorema 2.2 di pag. 32 del libro di testo, ovvero che il metodo di Newton converge linearmente verso una radice di molteplicità $m > 1$ con costante asintotica pari a:

$$\frac{(m-1)}{m}$$

La funzione data ha molteplicità della radice pari a 20, quindi la costante asintotica diventa:

$$\frac{(20-1)}{20} = \frac{19}{20}$$

che è coerente con quanto direttamente osservato dall'esecuzione del metodo.

Esercizio 2.5

Partendo da $x_0 = 3$, comparare (in termini di numero di iterazioni e di numero di valutazioni di funzione) il metodo di Newton, il metodo di Newton modificato e il metodo di accelerazione di Aitken (combinato a Newton) per approssimare le zero doppio $\hat{x} = 2$ della seguente funzione:

$$f(x) = 2^{x^2-4x+8} - 16$$

Verificare che in questo caso l'uso del criterio modificato per Newton non comporta variazioni sulle sue prestazioni. Sapresti motivarlo?

Svolgimento

$$x_0 = 3$$

<i>tolx</i>	<i>Aitken</i>	<i>Newton</i>	<i>Newton c</i>
10^{-1}	$\tilde{x} \approx 1.999$ $i = 3 \quad n = 12$	$\tilde{x} \approx 2.639$ $i = 1 \quad n = 2$	$\tilde{x} \approx 2.639$ $i = 1 \quad n = 2 \quad c \approx \dots$
10^{-2}	$\tilde{x} \approx 2 + 4 * 10^{-11}$ $i = 4 \quad n = 16$	$\tilde{x} \approx 2.02$ $i = 6 \quad n = 12$	$\tilde{x} \approx 2.02$ $i = 6 \quad n = 12 \quad c \approx 0.5$
10^{-3}	\vdots	$\tilde{x} \approx 2.002$ $i = 9 \quad n = 18$	$\tilde{x} \approx 2.002$ $i = 9 \quad n = 18 \quad c \approx 0.5$
10^{-4}	\vdots	$\tilde{x} \approx 2 + 1 * 10^{-4}$ $i = 13 \quad n = 26$	$\tilde{x} \approx 2 + 2 * 10^{-5}$ $i = 13 \quad n = 26 \quad c \approx 0.5$
10^{-5}	$\tilde{x} \approx 2 + 4 * 10^{-11}$ $i = 5 \quad n = 18$	$\tilde{x} \approx 2 + 2 * 10^{-5}$ $i = 16 \quad n = 32$	$\tilde{x} \approx 2 + 2 * 10^{-5}$ $i = 16 \quad n = 32 \quad c \approx 0.5$
10^{-6}	\vdots	$\tilde{x} \approx 2 + 2 * 10^{-6}$ $i = 19 \quad n = 38$	$\tilde{x} \approx 2 + 2 * 10^{-6}$ $i = 19 \quad n = 38 \quad c \approx 0.5$
10^{-7}	\vdots	$\tilde{x} \approx 2 + 1 * 10^{-7}$ $i = 23 \quad n = 46$	$\tilde{x} \approx 2 + 1 * 10^{-7}$ $i = 23 \quad n = 46 \quad c \approx 0.5$
10^{-8}	\vdots	$\tilde{x} \approx 2 + 2 * 10^{-8}$ $i = 26 \quad n = 52$	$\tilde{x} \approx 2 + 2 * 10^{-8}$ $i = 26 \quad n = 52 \quad c \approx 0.5$
10^{-9}	\vdots	$\tilde{x} \approx 2 + 6 * 10^{-9}$ $i = 27 \quad n = 56$	$\tilde{x} \approx 2 + 6 * 10^{-9}$ $i = 27 \quad n = 56 \quad c \approx 0.5$
10^{-10}	\vdots	\vdots	\vdots
<i>eps</i>	\vdots	\vdots	\vdots

Valgono qui le considerazioni fatte per l'esercizio precedente circa l'iterata in più di Aitken e i risultati sulla prima tolleranza dei metodi a convergenza lineare. Inoltre, per le caratteristiche della funzione data, anche qui vale il teorema 2.2 di pag. 32 del libro di testo. La funzione data ha molteplicità della radice pari a 2, quindi la costante asintotica diventa:

$$\frac{m-1}{m} \rightarrow \frac{(2-1)}{2} = \frac{1}{2}$$

Questo significa che il nostro controllo sull'incremento relativo modificato, ovvero:

$$\frac{|x_{i+1} - x_i|}{1 + |x_{i+1}|} \leq \frac{1-c}{c} \text{tolx}$$

diventa semplicemente:

$$\frac{|x_{i+1} - x_i|}{1 + |x_{i+1}|} \leq \frac{1 - \frac{1}{2}}{\frac{1}{2}} * tol x = 1 * tol x$$

ovvero come quello non modificato. Pertanto non si osservano variazioni sulle prestazioni.

Esercizio 2.6

Implementare in MATLAB (*functions*) il metodo delle secanti (criterio incremento) e il metodo delle corde (criterio incremento modificato).

Svolgimento

I codici sono forniti come al solito di parte grafica. Per quanto riguarda il metodo delle secanti, l'algoritmo esegue un passo di Newton fuori dal ciclo per poter implementare la formula. Per quanto riguarda le corde, l'algoritmo esegue due passi fuori dal ciclo per consentire la scelta della costante asintotica opzionale.

[Codice MATLAB 2.6.1](#) – Secanti

[Codice MATLAB 2.6.2](#) – Corde

Esercizio 2.7

Costruire una tabella in cui si comparano le prestazioni (in termini di numero di iterazioni e di numero di valutazioni di funzioni richieste) del metodo di Newton, delle secanti e delle corde per approssimare lo zero semplice appartenente all'intervallo $\left(-\frac{\pi}{2}, \frac{\pi}{2}\right)$ della seguente funzione:

$$f(x) = \sin(x) - 2\cos(x)$$

Partire da $x_0 = 0.1$ e utilizzare valori decrescenti di $tol x = 10^{-1}, \dots, 10^{-7}$. Verificare inoltre che il metodo delle corde non converge se si sceglie $x_0 = 0$, cosa fanno gli altri due metodi?

Svolgimento

Viene usato il metodo di Newton senza costante asintotica, quello delle secanti senza costante asintotica e quello delle corde con costante asintotica. Per tutti e tre i metodi viene usato il criterio d'arresto dell'incremento "relativo". La funzione ha radice in circa 1.071487

$$x_0 = 0.1$$

<i>tolx</i>	<i>Newton</i>	<i>Secanti</i>	<i>Corde</i>
10^{-1}	$x \approx 1.1072789$	$x \approx 1.1094866$	$x \approx 1.0942292$
	$i = 3 \quad n = 6$	$i = 3 \quad n = 4$	$i = 26 \quad n = 27$
10^{-2}	$x \approx 1.1071487$	$x \approx 1.1071479$	$x \approx 1.1057131$
	$i = 4 \quad n = 8$	$i = 4 \quad n = 5$	$i = 42 \quad n = 43$
10^{-3}	$x \approx 1.1071487$	$x \approx 1.1071487$	$x \approx 1.1072878$
	$i = 4 \quad n = 8$	$i = 5 \quad n = 6$	$i = 59 \quad n = 60$
10^{-4}	\vdots	\vdots	$x \approx 1.1071352$
			$i = 76 \quad n = 77$
10^{-5}	$x \approx 1.1071487$	\vdots	$x \approx 1.1071500$
	$i = 5 \quad n = 10$		$i = 93 \quad n = 94$
10^{-6}	\vdots	\vdots	$x \approx 1.1071485$
			$i = 110 \quad n = 111$
10^{-7}	\vdots	$x \approx 1.1071487$	$x \approx 1.1071487$
		$i = 6 \quad n = 7$	$i = 126 \quad n = 127$

Il più efficiente è ovviamente Newton, che ha convergenza quadratica, ma si comporta bene anche il metodo delle secanti. Vediamo che succede con $x_0 = 0$.

<i>tolx</i>	<i>Newton</i>	<i>Secanti</i>	<i>Corde</i>
10^{-1}	$x \approx 1.1071476$	$x \approx 1.1124526$	<i>Non converge</i>
	$i = 4 \quad n = 8$	$i = 3 \quad n = 4$	
10^{-2}	\vdots	$x \approx 1.1071476$	\vdots
		$i = 4 \quad n = 5$	
10^{-3}	$x \approx 1.1071487$	$x \approx 1.1071487$	\vdots
	$i = 5 \quad n = 10$	$i = 5 \quad n = 6$	
10^{-4}	\vdots	\vdots	\vdots
10^{-5}	\vdots	\vdots	\vdots
10^{-6}	\vdots	\vdots	\vdots
10^{-7}	$x \approx 1.1071487$	$x \approx 1.1071487$	\vdots
	$i = 6 \quad n = 12$	$i = 6 \quad n = 7$	

L'unico a non convergere è il metodo delle corde. Per capirne il perché ricorriamo ad alcune conseguenze del teorema del punto fisso, che ricordiamo brevemente: chiamiamo $\phi(x_k)$ la funzione d'iterazione, ovvero $x_{k+1} = \phi(x_k)$. Allora, per $k \rightarrow \infty$ avremo che $\hat{x} = \phi(\hat{x})$, dove \hat{x} viene definito come il "punto fisso" della funzione. Allora, se esiste un $\delta > 0$ t. c. $\forall x, y \in (\hat{x} - \delta, \hat{x} + \delta)$, risulta che $|\phi(x) - \phi(y)| \leq L|x - y|$, con $0 \leq L < 1$.

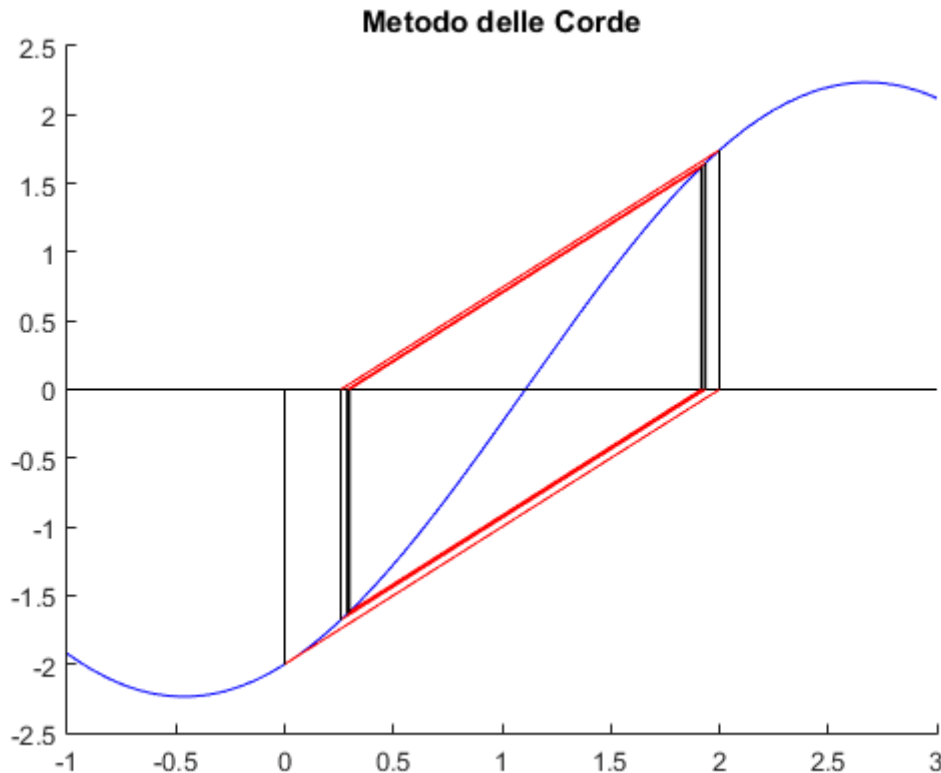
Sostanzialmente questo ci dice che, data $\phi \in C^1$, allora se $|\phi'(\hat{x})| < 1$ è possibile individuare un intorno in cui si va in convergenza. Quindi se $\phi'(\hat{x}) > 1$ o $\phi'(\hat{x}) < -1$ la funzione d'iterazione non converge. Nel nostro caso, ricordando che $x_0 = 0$, $\hat{x} \approx 1.071487$, abbiamo:

$$f(x) = \sin(x) - 2\cos(x), \quad f'(x) = 2\sin(x) + \cos(x)$$

$$\phi(x) = x - \frac{f(x)}{f'(x_0)} = x - \frac{\sin(x) - 2\cos(x)}{2\sin(x_0) + \cos(x_0)} = x - \sin(x) + 2\cos(x)$$

$$\phi'(\hat{x}) = 1 - \cos(\hat{x}) - 2\sin(\hat{x}) \approx -1.23456 < -1$$

Quando abbiamo $\phi'(\hat{x}) < -1$ quel che si verifica è che il metodo iterativo “gira intorno” alla radice, non andando mai in convergenza.



Esercizio 2.8

Completare la tabella realizzata al punto precedente inserendo nel confronto anche il metodo di bisezione scegliendo due diversi (ma sempre adeguati!) intervalli di confidenza iniziali.

Svolgimento

Gli intervalli di confidenza scelti sono $[0, 2]$ e $[-2, 4]$, due intervalli che verificano la condizione del teorema degli zeri prevista per il metodo di bisezione, ovvero $a, b \in \mathbb{R}$ t. c. $f(a)f(b) < 0$, che è equivalente a chiedere che lo zero della funzione si trovi all'interno dell'intervallo $[a, b]$. Inoltre, data la natura periodica della funzione data, questi due intervalli presentano un'unica radice, che è la stessa presente nell'intervallo dato nella traccia.

tolx	Newton	Secanti	Corde	Bisezione [0,2]	Bisezione [-2,4]
10^{-1}	$x \approx 1.1072789$ $i = 3 \quad n = 6$	$x \approx 1.1094866$ $i = 3 \quad n = 4$	$x \approx 1.0942292$ $i = 26 \quad n = 27$	$x = 1.125$ $i = 3 \quad n = 6$	$x = 1.1875$ $i = 4 \quad n = 7$
10^{-2}	$x \approx 1.1071487$ $i = 4 \quad n = 8$	$x \approx 1.1071479$ $i = 4 \quad n = 5$	$x \approx 1.1057131$ $i = 42 \quad n = 43$	$x = 1.109375$ $i = 6 \quad n = 9$	$x \approx 1.1054687$ $i = 8 \quad n = 11$
10^{-3}	$x \approx 1.1071487$ $i = 4 \quad n = 8$	$x \approx 1.1071487$ $i = 5 \quad n = 6$	$x \approx 1.1072878$ $i = 59 \quad n = 60$	$x \approx 1.1074218$ $i = 9 \quad n = 12$	$x \approx 1.1069335$ $i = 11 \quad n = 14$
10^{-4}	\vdots	\vdots	$x \approx 1.1071352$ $i = 76 \quad n = 77$	$x \approx 1.1071777$ $i = 12 \quad n = 15$	$x \approx 1.1071166$ $i = 14 \quad n = 17$
10^{-5}	$x \approx 1.1071487$ $i = 5 \quad n = 10$	\vdots	$x \approx 1.1071500$ $i = 93 \quad n = 94$	$x \approx 1.1071472$ $i = 15 \quad n = 18$	$x \approx 1.1071395$ $i = 17 \quad n = 20$
10^{-6}	\vdots	\vdots	$x \approx 1.1071485$ $i = 110 \quad n = 111$	$x \approx 1.1071491$ $i = 19 \quad n = 22$	$x \approx 1.1071481$ $i = 20 \quad n = 23$
10^{-7}	\vdots	$x \approx 1.1071487$ $i = 6 \quad n = 7$	$x \approx 1.1071487$ $i = 126 \quad n = 127$	$x \approx 1.1071486$ $i = 21 \quad n = 24$	$x \approx 1.1071487$ $i = 24 \quad n = 27$

Codice MATLAB 2.8 – Bisezione

ESERCIZI CAPITOLO 3

SISTEMI LINEARI E NON LINEARI

Esercizio 3.1

Implementare in MATLAB una procedura per risolvere un sistema lineare triangolare inferiore a diagonale unitaria e testarne il funzionamento.

Svolgimento

Per la risoluzione di questo esercizio presentiamo due versioni dell'algoritmo particolarmente attente ai criteri di robustezza ed efficienza. La prima versione è caratterizzata da codice che tiene conto del verificarsi di situazioni di errore, ovvero quando la matrice dei coefficienti passata in argomento alla funzione non è quadrata oppure non è a diagonale unitaria. L'algoritmo prende spunto dal 3.1 di pag. 46 del libro di testo, con l'osservazione che non occorre eseguire la divisione ogni volta che si esce dal ciclo più interno dato che tale operazione consisterebbe nel dividere un numero per 1. Nella prima versione dell'algoritmo, viene inizialmente controllato se la matrice è quadrata ed iterativamente se la sua diagonale è unitaria. La seconda versione dell'algoritmo invece è quella che verrà richiamata negli esercizi successivi ed è priva dei controlli su menzionati poiché vengono effettuati da altre funzioni, rendendo l'algoritmo più veloce. Entrambe le implementazioni effettuano l'accesso alla matrice per righe sfruttando una caratteristica importante del linguaggio MATLAB, ovvero che le matrici vengono memorizzate in memoria in indirizzi contigui, come se fossero degli array.

[Codice MATLAB 3.1.1](#) (con controlli)

[Codice MATLAB 3.1.2](#) (senza controlli)

Test di funzionamento.

Risolvendo il sistema triangolare inferiore a diagonale unitaria $A\underline{x} = \underline{b}$ con matrice

$$A = \begin{pmatrix} 1 & 0 & 0 \\ 3 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix} \text{ e } b = \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix}, \text{ otteniamo:}$$

```
Command Window
4
-7
9

>> A\b

ans =

4
-7
9
```

Test di non funzionamento.

Data la matrice $A = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}$ otteniamo:

```
Command Window
Error using Ex_3_1 (line 15)
La matrice non è quadrata
```

Data la matrice $A = \begin{pmatrix} 2 & 2 \\ 3 & 4 \end{pmatrix}$ otteniamo:

```
Command Window
Error using Ex_3_1 (line 23)
La matrice non è a diagonale unitaria
```

Esercizio 3.2

Scrivere una *function* MATLAB che risolve efficientemente $K \geq 1$ sistemi lineari quadrati $Ax = b_j, j = 1, \dots, K$ (stessa matrice dei coefficienti). A tale scopo richiamare innanzitutto la *function* MATLAB che riscrive la matrice A con l'algoritmo 3.5 e richiamare poi le *functions* opportune per la risoluzione dei sistemi triangolari. Testarne quindi il funzionamento usando una matrice A a diagonale dominante per righe o per colonne e il non funzionamento usando una matrice nonsingolare ma con il minore principale di ordine 2 nullo.

Svolgimento

Per la risoluzione di questo esercizio è stato implementato l'algoritmo di fattorizzazione richiesto aggiungendovi un controllo sulle dimensioni della matrice, in modo che vengano accettate solo matrici quadrate. La *function* richiama quindi l'algoritmo di fattorizzazione LU e, utilizzando il codice 3.1.2 (triangolare inferiore) e l'algoritmo 3.3 del libro di testo (triangolare superiore), risolve k sistemi lineari e restituisce il vettore soluzione dei sistemi

presi in considerazione, che vengono memorizzati nella stessa matrice nella quale erano memorizzati i termini noti.

[Codice MATLAB 3.2.1](#) (codice che risolve k sistemi)

[Codice MATLAB 3.2.2](#) (codice per la fattorizzazione LU)

[Codice MATLAB 3.2.3](#) (codice che risolve un sistema triangolare superiore)

Test di funzionamento.

Risolviamo $k = 2$ sistemi con:

$$A = \begin{pmatrix} 10 & 2 & 1 \\ 2 & 20 & 3 \\ 1 & 3 & 30 \end{pmatrix} \text{ e } B = \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}$$

L'esecuzione del nostro metodo produce i seguenti risultati:

```
Command Window
    0.0752    0.3450
    0.0790    0.1901
    0.0896    0.1695
fx >> |
```

Come riprova utilizziamo il comando per risolvere sistemi di Matlab:

```
Command Window
>> A\B
ans =
    0.0752    0.3450
    0.0790    0.1901
    0.0896    0.1695
fx >> |
```

Test di non funzionamento.

Data la matrice $A = \begin{pmatrix} 2 & 3 & 3 \\ 6 & 9 & 3 \\ 1 & 1 & 1 \end{pmatrix}$ con minore principale di ordine 2 nullo otteniamo:

```
Command Window
Error using fattorizzazioneLU (line 18)
La matrice non è fattorizzabile LU
```

Data la matrice $A = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}$ otteniamo:

Command Window

Error using `fattorizzazioneLU` (line 14)
La matrice non è quadrata

Esercizio 3.3

Scrivere una *function* MATLAB che risolve efficientemente $K \geq 1$ sistemi lineari quadrati $Ax = b_j$, $j = 1, \dots, K$ con A simmetrica. A tale scopo richiamare innanzitutto la *function* MATLAB che riscrive la matrice A con l'algoritmo 3.6 e richiamare poi le *functions* opportune per la risoluzione dei sistemi triangolari (risolvere il sistema diagonale all'interno di questa *function*). Testarne quindi il funzionamento usando una matrice A *sdp* e il non funzionamento per una matrice simmetrica ma non definita positiva (mettere un elemento sulla diagonale negativo).

Svolgimento

Per la risoluzione di questo esercizio è stato implementato l'algoritmo di fattorizzazione richiesto aggiungendovi un controllo sulle dimensioni della matrice e sugli elementi della diagonale principale, in modo che vengano accettate solo matrici quadrate e *sdp*. La *function* richiama quindi l'algoritmo di fattorizzazione LDL^T e, attraverso il codice 3.1.2 e ad altri due algoritmi da noi implementati, riesce a risolvere k sistemi lineari ed a restituire i vettori soluzione dei sistemi presi in considerazione che vengono memorizzati nella stessa matrice dei termini noti. Per risolvere i sistemi lineari indotti dalla fattorizzazione LDL^T abbiamo implementato un algoritmo che risolve un sistema diagonale ed un altro che risolve un sistema triangolare superiore a diagonale unitaria. Per quest'ultimo valgono le stesse considerazioni viste in precedenza per il codice 3.1.2.

[Codice MATLAB 3.3.1](#) (codice che risolve k sistemi)

[Codice MATLAB 3.3.2](#) (codice per la fattorizzazione LDL^T)

[Codice MATLAB 3.3.3](#) (codice che risolve un sistema diagonale)

[Codice MATLAB 3.3.4](#) (codice che risolve un sistema triangolare superiore a diagonale unitaria)

Test di funzionamento.

Con matrice *sdp* $A = \begin{pmatrix} 4 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 8 \end{pmatrix}$ e $B = \begin{pmatrix} 1 & 1 \\ 0 & 2 \\ 1 & -1 \end{pmatrix}$

eseguendo il nostro metodo otteniamo:

```
Command Window

    0.3333    -0.3333
   -0.3333     2.3333
    0.1250    -0.1250

fx >> |
```

Come riprova utilizziamo il comando per risolvere sistemi di Matlab:

```
Command Window

>> A\B

ans =

    0.3333    -0.3333
   -0.3333     2.3333
    0.1250    -0.1250

fx >> |
```

Test di non funzionamento.

Con la matrice $A = \begin{pmatrix} 3 & 6 & 9 \\ 6 & -1 & 1 \\ 9 & 1 & 3 \end{pmatrix}$ otteniamo:

```
Command Window

Error using fattorizzazioneLDLT (line 29)
La matrice non è SDP
```

Con la matrice $A = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}$ otteniamo:

```
Command Window

Error using fattorizzazioneLDLT (line 14)
La matrice non è quadrata
```

Esercizio 3.4

Scrivere una *function* MATLAB che risolve efficientemente $K \geq 1$ sistemi lineari quadrati $Ax = b_j$, $j = 1, \dots, K$ (stessa matrice dei coefficienti). A tale scopo richiamare innanzitutto la *function* MATLAB che riscrive la matrice A con l'algoritmo 3.7 e produce il vettore di permutazione p . Poi, permutati i termini noti, richiamare le *functions* opportune per la risoluzione dei sistemi triangolari. Testarne quindi il funzionamento usando una matrice A nonsingolare ma con il minore principale di ordine 2 nullo.

Svolgimento

Per la risoluzione di questo esercizio è stato implementato l'algoritmo di fattorizzazione richiesto aggiungendovi due controlli in modo da fattorizzare solo matrici quadrate e nonsingolari. La *function* richiama quindi l'algoritmo di fattorizzazione LU con pivoting parziale, che oltre a restituire la matrice fattorizzata produrrà anche il vettore di permutazione come richiesto nella traccia dell'esercizio. Attraverso il codice 3.1.2 e all'algoritmo 3.3 del libro di testo a pag. 47, riesce a risolvere k sistemi lineari ed a restituire i vettori soluzione dei sistemi presi in considerazione che vengono memorizzati nella stessa matrice nella quale erano memorizzati i termini noti.

[Codice MATLAB 3.4.1](#) (codice che risolve k sistemi)

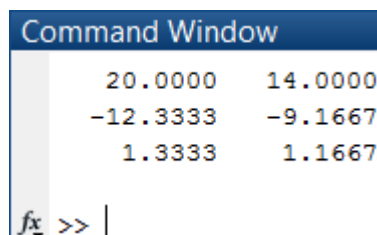
[Codice MATLAB 3.4.2](#) (codice per la fattorizzazione LU con pivoting parziale)

Test di funzionamento.

Prendiamo la stessa matrice usata nel 3.2:

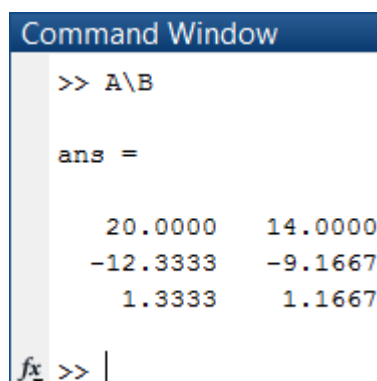
$$A = \begin{pmatrix} 2 & 3 & 3 \\ 6 & 9 & 3 \\ 1 & 1 & 1 \end{pmatrix} \text{ e } B = \begin{pmatrix} 7 & 4 \\ 13 & 5 \\ 9 & 6 \end{pmatrix}.$$

L'esecuzione del nostro codice porta ad avere:



```
Command Window
    20.0000    14.0000
   -12.3333    -9.1667
     1.3333     1.1667
fx >> |
```

Come riprova utilizziamo il comando MATLAB per risolvere sistemi:



```
Command Window
>> A\b
ans =
    20.0000    14.0000
   -12.3333    -9.1667
     1.3333     1.1667
fx >> |
```

Test di non funzionamento.

Con la matrice $A = \begin{pmatrix} 12 & 2 & 3 \\ 2 & 5 & 6 \end{pmatrix}$

otteniamo:

Command Window

```
Error using fattorizzazioneLUpivoting (line 16)  
La matrice non è quadrata
```

Con la matrice $A = \begin{pmatrix} 1 & 0 & 3 \\ 0 & 0 & 3 \\ 0 & 0 & 0 \end{pmatrix}$ singolare, otteniamo:

Command Window

```
Error using fattorizzazioneLUpivoting (line 26)  
La matrice è singolare
```

Esercizio 3.5 (facoltativo)

Algoritmi per sistemi lineari in aritmetica finita. Usando un'aritmetica finita che lavora in base 10 con arrotondamento e usando $m = 3$ cifre per la mantissa normalizzata (e s cifre per l'esponente e uno *shift* v tali che non si abbiano mai problemi di underflow né di overflow nello svolgimento dell'esercizio), risolvere il sistema lineare $Ax = b$, con

$$A = \begin{pmatrix} 10^{-4} & 1 \\ 1 & 1 \end{pmatrix}, \quad b = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

A tale scopo calcolare in tale aritmetica la fattorizzazione LU della matrice e risolvere i sistemi triangolari. Successivamente scambiare la prima riga di A e di b con le loro rispettive seconde righe e ripetere il calcolo. Confrontare infine l'accuratezza delle due soluzioni così ottenute (calcolare l'errore relativo per ciascuna delle due componenti della soluzione esatta).

Svolgimento

Risolviamo innanzitutto il sistema in aritmetica esatta:

$$\begin{pmatrix} 10^{-4} & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \end{pmatrix} = \begin{cases} 10^{-4}x_1 + x_2 = 1 \\ x_1 + x_2 = 2 \end{cases}$$

$$x_1 = \frac{10000}{9999}$$

$$x_2 = \frac{9998}{9999}$$

Risolviamo ora in aritmetica finita il sistema mediante fattorizzazione LU . I numeri di partenza del sistema, con le condizioni imposte dalla traccia, sono esattamente rappresentabili. Partiamo con la costruzione del vettore di Gauss:

$$g_1 = \frac{1}{10^{-4}} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 10^4 \end{pmatrix}$$

Questa operazione non ci dà problemi. Moltiplichiamo ora tale vettore per e^T :

$$g_1 e_1^T = \begin{pmatrix} 0 \\ 10^4 \end{pmatrix} (1 \ 0) = \begin{pmatrix} 0 & 0 \\ 10^4 & 0 \end{pmatrix}$$

Anche questa operazione non è problematica. Adesso otteniamo L_1 :

$$L_1 = I \ominus g_1 e_1^T = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \ominus \begin{pmatrix} 0 & 0 \\ 10^4 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ -10^4 & 1 \end{pmatrix}$$

Anche qui, nessun problema. Il primo errore di rappresentazione si riscontra nel calcolo di $A_2 \equiv U$:

$$U \equiv A_2 = L_1 A_1 = \begin{pmatrix} 1 & 0 \\ -10^4 & 1 \end{pmatrix} \begin{pmatrix} 10^{-4} & 1 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} 10^{-4} & 1 \\ 0 & -10^4 \end{pmatrix}$$

Il termine evidenziato in rosso dovrebbe essere $1 - 10^4 = -9999$, ma è soggetto ad errore di rappresentazione. Normalizzato, il numero diventa $-9.999 * 10^3$. Avendo a disposizione solo 3 bit di mantissa, e sapendo che nel caso di arrotondamento α_m diventa $\alpha_m + 1$ se $\alpha_{m+1} \geq \frac{b}{2}$, la floating point di tale numero è pari a -10^4 .

Andiamo ora a risolvere il sistema:

$$\begin{cases} Ly = b \\ Ux = y \end{cases}$$

$$Ly = b \rightarrow \begin{pmatrix} 1 & 0 \\ 10^4 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \end{pmatrix} \rightarrow \begin{cases} y_1 = 1 \\ y_2 = 2 \ominus 10^4 = -10^4 \end{cases}$$

Anche qui riscontriamo un errore di rappresentazione. Continuando lo svolgimento del sistema abbiamo:

$$Ux = y \rightarrow \begin{pmatrix} 10^{-4} & 1 \\ 0 & -10^4 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ -10^4 \end{pmatrix} \rightarrow \begin{cases} 10^{-4}x_1 \oplus x_2 = 1 \\ -10^4x_2 = -10^4 \end{cases} \rightarrow \begin{cases} x_1 = 0 \\ x_2 = 1 \end{cases}$$

Calcoliamo ora l'errore relativo per ciascuna delle due componenti della soluzione esatta:

$$|\mathcal{E}^{(1)}_{x_1}| = \left| \frac{\tilde{x}_1 - x_1}{x_1} \right| = \left| \frac{\frac{10000}{9999} - 0}{\frac{10000}{9999}} \right| = 1$$

$$|\mathcal{E}^{(1)}_{x_2}| = \left| \frac{\tilde{x}_2 - x_2}{x_2} \right| = \left| \frac{\frac{9998}{9999} - 1}{\frac{9998}{9999}} \right| = \frac{1}{9998} \approx 10^{-4}$$

Notiamo come per la componente x_1 si abbia una totale perdita d'informazione.

Risolviamo ora in aritmetica finita il sistema con la permutazione indicata nella traccia:

$$\begin{pmatrix} 1 & 1 \\ 10^{-4} & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \end{pmatrix}$$

Calcoliamo il vettore di Gauss:

$$g_1 = 1 \begin{pmatrix} 0 \\ 10^{-4} \end{pmatrix} = \begin{pmatrix} 0 \\ 10^{-4} \end{pmatrix}$$

Questa operazione non ci dà problemi. Moltiplichiamo ora tale vettore per e^T :

$$g_1 e_1^T = \begin{pmatrix} 0 \\ 10^{-4} \end{pmatrix} (1 \quad 0) = \begin{pmatrix} 0 & 0 \\ 10^{-4} & 0 \end{pmatrix}$$

Anche questa operazione non è problematica. Adesso otteniamo L_1 :

$$L_1 = I \ominus g_1 e_1^T = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \ominus \begin{pmatrix} 0 & 0 \\ 10^{-4} & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ -10^{-4} & 1 \end{pmatrix}$$

Anche qui, nessun problema. Analogamente a quanto visto in precedenza, il primo errore di rappresentazione si riscontra nel calcolo di $A_2 \equiv U$:

$$U \equiv A_2 = L_1 A_1 = \begin{pmatrix} 1 & 0 \\ -10^{-4} & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 10^{-4} & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 0 & \textcolor{red}{1} \end{pmatrix}$$

Il termine evidenziato in rosso dovrebbe essere $1 - 10^{-4} = 0.9999$, ma è soggetto ad errore di rappresentazione. Normalizzato, il numero diventa $9.999 \cdot 10^{-1}$. In questa aritmetica finita, la floating point di tale numero è pari a 1.

Andiamo ora a risolvere il sistema:

$$\begin{cases} Ly = b \\ Ux = y \end{cases}$$

$$Ly = b \rightarrow \begin{pmatrix} 1 & 0 \\ 10^{-4} & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \end{pmatrix} \rightarrow \begin{cases} y_1 = 2 \\ y_2 = 1 \ominus 2 \otimes 10^{-4} = 1 \ominus \textcolor{red}{0} = 1 \end{cases}$$

Stavolta l'errore è sull'operazione di moltiplicazione: $2 \cdot 10^{-4} = 0.0002$, la cui floating point in quest'aritmetica finita risulta 0. Continuando lo svolgimento del sistema abbiamo:

$$Ux = y \rightarrow \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \end{pmatrix} \rightarrow \begin{cases} x_1 \oplus x_2 = 2 \\ x_2 = 1 \end{cases} \rightarrow \begin{cases} x_1 = 1 \\ x_2 = 1 \end{cases}$$

Calcoliamo ora l'errore relativo per ciascuna delle due componenti della soluzione esatta:

$$|\varepsilon^{(2)}_{x_1}| = \left| \frac{\tilde{x}_1 - x_1}{x_1} \right| = \left| \frac{\frac{10000}{9999} - 1}{\frac{10000}{9999}} \right| = \frac{1}{10000} = 10^{-4}$$

$$|\varepsilon^{(2)}_{x_2}| = \left| \frac{\tilde{x}_2 - x_2}{x_2} \right| = \left| \frac{\frac{9998}{9999} - 1}{\frac{9998}{9999}} \right| = \frac{1}{9998} \approx 10^{-4}$$

Possiamo finalmente confrontare l'accuratezza delle due soluzioni: per quanto riguarda la componente x_1 , nel primo sistema abbiamo una totale perdita d'informazione, mentre nel secondo è pressoché nulla ($\approx 0.01\%$). Per quanto riguarda la componente x_2 abbiamo la

stessa perdita d'informazione per entrambi i sistemi ($\approx 0.01\%$). Si evince che il sistema permutato soffre meno gli errori dovuti all'aritmetica finita. Osserviamo che, in generale, scegliere il pivot come l'elemento massimo in valore assoluto della colonna ad ogni iterazione della fattorizzazione (che è l'idea della fattorizzazione con pivoting parziale), migliora la stabilità dell'algoritmo e l'accuratezza del risultato ottenuto.

Esercizio 3.6

Scrivere una *function* MATLAB che implementa il metodo di Newton per approssimare uno zero di una funzione $F: \mathbb{R}^2 \rightarrow \mathbb{R}^2$. A tale scopo ad ogni iterazione del metodo richiamare la *function* costruita per l'Esercizio 4.

Svolgimento

Il problema che vogliamo risolvere è:

$$F(y) = 0, \quad F: \Omega \subseteq \mathbb{R}^n \rightarrow \mathbb{R}^n$$

Definiamo formalmente il metodo di Newton come:

$$y^{k+1} = y^k - J_F(y^k)^{-1}F(y^k), \quad k = 0, 1, \dots$$

a partire da un'approssimazione iniziale y^0 , dove $J_F(y)$ è il Jacobiano di F . Ad ogni passo andiamo a risolvere il sistema:

$$\text{per } k = 0, 1, \dots$$

$$J_F(y^k)d^k = -F(y^k)$$

$$y^{k+1} = y^k + d^k$$

Iterativamente risolviamo una successione di sistemi lineari richiamando l'algoritmo di fattorizzazione *LU* con pivoting parziale vista la necessità di fattorizzare la matrice Jacobiana ad ogni passo. Se il punto d'innescio è scelto opportunamente in un intorno della soluzione esatta, il metodo di Newton converge quadraticamente. Ad ogni modo, abbiamo introdotto un numero di iterazioni massime entro il quale far terminare l'algoritmo.

[Codice MATLAB 3.6](#)

Esercizio 3.7

Calcolare analiticamente i due punti di intersezione fra l'ellisse $4x_1^2 + x_2^2 = 4$ e la parabola $x_2 = x_1^2 - 1$. Testare il funzionamento della *function* MATLAB costruita al punto precedente interpretando tali punti come zeri di una funzione non lineare $F: \mathbb{R}^2 \rightarrow \mathbb{R}^2$ (selezionare per ciascuno di essi un opportuno punto di innescio per il metodo di Newton).

Svolgimento

Calcoliamo analiticamente i due punti di intersezione tra l'ellisse $4x_1^2 + x_2^2 = 4$ e la parabola $x_2 = x_1^2 - 1$:

$$\begin{cases} x_1^2 - x_2 = 1 \\ 4x_1^2 + x_2^2 = 4 \end{cases} \rightarrow \begin{cases} x_2 = -1 + x_1^2 \\ 4x_1^2 + (-1 + x_1^2)^2 = 4 \end{cases} \rightarrow \begin{cases} x_2 = -1 + x_1^2 \\ 4x_1^2 + 1 + x_1^4 - 2x_1^2 = 4 \end{cases}$$

Poniamo $z = x_1^2$ allo scopo di risolvere l'equazione biquadratica, ottenendo quindi:

$$z^2 + 2z - 3 = 0$$

che ha come uniche radici reali $x_1 = \pm 1$. Andiamo a sostituire alla prima equazione i valori di x_1 , ottenendo in entrambi i casi:

$$1 - x_2 = 1 \rightarrow x_2 = 0$$

Pertanto i punti d'intersezione sono:

$$P_1 = (1,0)$$

$$P_2 = (-1,0)$$

Abbiamo verificato sperimentalmente che, scegliendo (10,10) come punto d'innescio, il metodo di Newton va in convergenza su P_1 , scegliendo invece (-10,10) il metodo converge su P_2 .

Esercizio 3.8

Scrivere una *function* MATLAB che risolve nel senso dei minimi quadrati $K \geq 1$ sistemi lineari sovradeterminati $Ax = b_j$, $j = 1, \dots, K$ (dove A è $m \times n$, $m > n = \text{rank}(A)$). A tale scopo richiamare innanzitutto la *function* MATLAB che riscrive la matrice A con l'algoritmo 3.8 e calcolare efficientemente i vettori $Q^T b_j$, $j = 1, \dots, K$ usando tale matrice riscritta. Infine richiamare la *function* per la risoluzione di un sistema lineare triangolare superiore. Testarne il funzionamento su una matrice 4×2 di rango 2 (confrontare la soluzione con quella che si ottiene con il comando $A \backslash b_j$) e il non funzionamento su una matrice delle stesse dimensioni ma di rango 1.

Svolgimento

Per la risoluzione di questo esercizio è stato implementato l'algoritmo di fattorizzazione richiesto aggiungendovi due controlli in modo da fattorizzare solo matrici nonsingolari con numero di righe maggiore del numero delle colonne. In particolare è stato implementato un controllo sul valore di α diverso da quello presente nel codice 3.8 del libro di testo a pag. 75: sfruttando il fatto che la norma euclidea non varia per trasformazioni ortogonali, abbiamo calcolato tale valore fuori dal ciclo salvandolo in una variabile d'appoggio. Inoltre si è utilizzata un'altra variabile chiamata *cost* per regolare la tolleranza del controllo. Possiamo

osservare che già da $cost = 1$ il controllo risulta robusto anche contro errori di aritmetica finita che erano stati riscontrati con alcune matrici, com'è evidenziato nei test di non funzionamento. La *function* richiama quindi l'algoritmo di fattorizzazione QR, che oltre a restituire la matrice fattorizzata, produrrà anche il vettore residuo che risulterà utile per gli esercizi del capitolo 4. Attraverso l'algoritmo 3.3 del libro vengono risolti k sistemi lineari e vengono restituiti i vettori soluzione dei sistemi presi in considerazione che vengono memorizzati nella stessa matrice nella quale erano memorizzati i termini noti, oltre alla restituzione del vettore residuo come sopra specificato.

[Codice MATLAB 3.8.1](#) (codice che risolve k sistemi)

[Codice MATLAB 3.8.2](#) (codice per la fattorizzazione QR)

Test di funzionamento.

$$A = \begin{pmatrix} 4 & 1 \\ 7 & 2 \\ 8 & 2 \\ 14 & 4 \end{pmatrix} \text{ e } B = \begin{pmatrix} 1 & 2 \\ 1 & 1 \\ 1 & 1 \\ 2 & 1 \end{pmatrix}$$

Mettiamo a confronto i nostri risultati e quelli di MATLAB:

Command Window	
0.2000	1.0000
-0.2000	-3.2000

Command Window	
>> A\B	
ans =	
0.2000	1.0000
-0.2000	-3.2000
fx >>	

Test di non funzionamento.

Con la matrice $A = \begin{pmatrix} 2 & 1 \\ 4 & 2 \\ 8 & 4 \\ 16 & 8 \end{pmatrix}$ otteniamo:

Command Window	
Error using <u>fattorizzazioneQR</u> (line 25)	
La matrice non ha rango massimo	

Con la matrice $A = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}$ otteniamo:

Command Window	
Error using <u>fattorizzazioneQR</u> (line 14)	
Il numero di righe deve essere strettamente maggiore del numero delle colonne	

ESERCIZI CAPITOLO 4

APPROSSIMAZIONE DI FUNZIONI

Esercizio 4.1

Sia $f(x) = 4x^2 - 12x + 1$. Determinare $p(x) \in \Pi_4$ che interpola $f(x)$ sulle ascisse $x_i = i$, $i = 0, \dots, 4$.

Svolgimento

Osserviamo che si chiede di determinare il polinomio interpolante di quarto grado di una funzione che è già un polinomio, il cui grado è minore di 4. Pertanto il polinomio interpolante la funzione sarà la funzione stessa. Possiamo anche verificarlo utilizzando il polinomio interpolante di Newton:

x	0	1	2	3	4
f	1	-7	-7	1	17

$$w_0(x) = 1$$

$$w_1(x) = x$$

$$w_2(x) = x(x - 1)$$

$$w_3(x) = x(x - 1)(x - 2)$$

$$w_4(x) = x(x - 1)(x - 2)(x - 3)$$

Utilizzando l'algoritmo 4.1 per ricavare i valori delle differenze divise, otteniamo la seguente diagonale: 1, -8, 4, 0, 0. Coerentemente con quanto osservato in precedenza, gli ultimi due termini sono nulli. Svolgendo i conti, si torna alla funzione polinomiale di partenza:

$$1 - 8x + 4x(x - 1) = 1 - 8x + 4x^2 - 4x = 4x^2 - 12x + 1$$

Esercizio 4.2

Dimostrare che il seguente algoritmo,

```
p = a(n + 1)
for k = n:-1:1
    p = p * x + a(k)
end
```

valuta il polinomio

$$p(x) = \sum_{k=0}^n a_k x^k$$

nel punto x , se il vettore \underline{a} contiene i coefficienti del polinomio $p(x)$. (Osservare che in MATLAB i vettori hanno indice che parte da 1, invece che da 0).

Svolgimento

La prerogativa sulla quale si basa l'algoritmo è che $x^k = x^{k-1} x$. Osserviamo quindi che è possibile riscrivere il $p(x)$ come segue:

$$\begin{aligned} p(x) &= \sum_{k=0}^n a_k x^k = a_0 + a_1 x + a_2 x^2 + \dots + a_{n-1} x^{n-1} + a_n x^n \\ &= a_0 + x(a_1 + x(a_2 + \dots + x(a_{n-1} + a_n x) \dots)) \end{aligned}$$

In quest'ultima forma vengono svolte n addizioni e n moltiplicazioni, anziché le n addizioni e le $\frac{n(n+1)}{2}$ moltiplicazioni della forma classica. Pertanto, svolgendo ad ogni passo un'addizione ed un prodotto, si ha un costo di $2n$ flops. Avendo a disposizione il vettore dei coefficienti \underline{a} , l'algoritmo di Horner valuta correttamente il polinomio in un dato punto x .

Esercizio 4.3

Dimostrare il lemma 4.1.

Svolgimento

La prima parte del lemma è:

dati i polinomi di Lagrange

$$L_{kn}(x) = \prod_{j=0, j \neq k}^n \frac{x - x_j}{x_k - x_j} \quad \text{con } k = 0, 1, \dots, n$$

definiti sulle ascisse $a \leq x_0 < x_1 < \dots < x_n \leq b$

abbiamo:

$$L_{kn}(x_i) = \begin{cases} 1, & \text{se } k = i \\ 0, & \text{se } k \neq i \end{cases}$$

Dimostrazione.

Si distinguono due casi:

- Se $k = i$, il termine $L_{kn}(x_i)$ avrà numeratore uguale al denominatore, quindi avremo:

$$L_{in}(x_i) = \prod_{j=0, j \neq i}^n \frac{x_i - x_j}{x_i - x_j} = 1$$

- Se $k \neq i$, $\exists j = k$ t.c. $x_k - x_j = 0$, e questo termine annullerà la produttoria.

La seconda parte del lemma è: i polinomi di Lagrange ha grado esatto n ed il coefficiente principale di $L_{kn}(x)$ è:

$$c_{kn} = \frac{1}{\prod_{j=0, j \neq k}^n (x_k - x_j)} \quad \text{con } k = 0, 1, \dots, n$$

Dimostrazione.

Osserviamo che ogni singolo termine che compone la produttoria di $L_{kn}(x)$ è di grado 1, pertanto $L_{kn}(x)$ ha grado n . La formula del coefficiente principale si ottiene banalmente per costruzione.

La terza parte del lemma è: i polinomi di Lagrange sono linearmente indipendenti tra loro; costituiscono, pertanto, una base per Π_n .

Dimostrazione.

Il sistema:

$$\sum_{k=0}^n c_k L_{kn}(x) = 0$$

è verificato se $c_k = 0 \forall k \in \mathbb{R}$ con $k = 0, \dots, n$. Considerando che $L_{kn}(x_k) = 1$ per quanto dimostrato in precedenza, allora dovrà essere $c_k = 0$. Dato che sono n polinomi, essi costituiscono una base per Π_n .

Esercizio 4.4

Dimostrare il lemma 4.2.

Svolgimento

Enunciato:

data come base di rappresentazione per Π_n la base di Newton

$$w_0(x) = 1$$

$$w_{k+1}(x) = (x - x_k) w_k(x)$$

per ogni $k = 0, 1, 2, \dots$ si ha che:

- $w_k(x) \in \Pi'_k$
- $w_{k+1}(x) = \prod_{j=0}^k (x - x_j)$
- $w_{k+1}(x_j) = 0, \text{ per } j \leq k$
- $w_0(x), \dots, w_k(x)$ costituiscono una base per Π_n

Dimostrazioni:

- Si dimostra per induzione. Caso base: $w_0(x) = 1$ quindi $w_0(x) \in \Pi'_0$. Supponiamo la tesi vera per k , dimostriamo che $w_{k+1}(x) = (x - x_k) w_k(x) \in \Pi'_{k+1}$. È banalmente dimostrato in quanto per ipotesi induttiva $w_k \in \Pi'_k$ e $(x - x_k) \in \Pi'_1$.
- $w_{k+1}(x) = (x - x_k)w_k = (x - x_k)(x - x_{k-1})w_{k-1}(x) = \dots = \prod_{j=0}^k (x - x_j)$.
- Per $j \leq k$, un fattore della produttoria, considerando la seconda proprietà, sarà $(x_j - x_j)$ quindi $w_{k+1}(x_j) = 0$.
- $c_i w_i(x_j) = 0$ implica che $c_i = 0$ per $j \neq i$. Quindi, dato che sono k , anche essi costituiscono una base per Π_k .

Esercizio 4.5

Dimostrare il teorema 4.4.

Svolgimento

Enunciato: valgono le seguenti proprietà delle differenze divise:

- Se $\alpha, \beta \in \mathbb{R}$ e $f(x), g(x)$ sono funzioni di una variabile reale,
 $(\alpha f + \beta g)[x_0, \dots, x_r] = \alpha f[x_0, \dots, x_r] + \beta g[x_0, \dots, x_r]$
- Per ogni $\{i_0, i_1, \dots, i_r\}$ permutazione di $\{0, 1, \dots, r\}$, $f[x_{i_0}, \dots, x_{i_r}] = f[x_0, \dots, x_r]$
- Sia $f(x) = \sum_{i=0}^k a_i x^i \in \Pi_k$, allora, $f[x_0, x_1, \dots, x_r] = \begin{cases} a_k, & \text{se } r = k \\ 0 & \text{se } r > k \end{cases}$
- Più in generale, se $f(x) \in C^{(r)}$, allora, $f[x_0, x_1, \dots, x_r] = \frac{f^{(r)}(\xi)}{r!}$ con $\xi \in [\min_i x_i, \max_i x_i]$
- $f[x_0, x_1, \dots, x_{r-1}, x_r] = \frac{f[x_1, \dots, x_r] - f[x_0, \dots, x_{r-1}]}{x_r - x_0}$

Dimostrazioni:

- $(\alpha f + \beta g)[x_0, \dots, x_r] = \sum_{k=0}^r \frac{\alpha f_k + \beta g_k}{\prod_{j=0, j \neq k}^r x_k - x_j} = \sum_{k=0}^r \frac{\alpha f_k}{\prod_{j=0, j \neq k}^r x_k - x_j} + \sum_{k=0}^r \frac{\beta g_k}{\prod_{j=0, j \neq k}^r x_k - x_j} =$
 $\alpha \sum_{k=0}^r \frac{f_k}{\prod_{j=0, j \neq k}^r x_k - x_j} + \beta \sum_{k=0}^r \frac{g_k}{\prod_{j=0, j \neq k}^r x_k - x_j} = \alpha f[x_0, \dots, x_r] + \beta g[x_0, \dots, x_r]$
- L'addizione ed il prodotto godono della proprietà commutativa, quindi l'ordine degli indici in una differenza divisa non ha importanza.
- Data l'unicità del polinomio interpolante, abbiamo che: $f(x) = \sum_{i=0}^k a_i x^i = \sum_{i=0}^k f[x_0, \dots, x_i] w_i(x) \equiv p_k(x)$
 Se consideriamo il k -esimo termine, dato che $w_k(x) \in \Pi'_k$, allora $f[x_0, \dots, x_k] = a_k$. I termini con $r > k$ non compaiono nella formula, pertanto le differenze divise sono nulle.
- Sia $p(x) = \sum_{i=0}^r w_i(x) f[x_0, \dots, x_i]$. Data l'espressione dell'errore $e(x) = f(x) - p(x) \in C^{(r)}$ in $[a, b]$, sappiamo che $e(x_i) = f(x_i) - p(x_i) = 0$ per ogni $i = 0, \dots, r$. Per il teorema di Rolle, $\exists \xi_i^{(1)} \in [x_i, x_{i+1}]$ t. c. $e'(\xi_i^{(1)}) = 0$ per $i = 0, \dots, r-1$. Riapplicando il

teorema di Rolle, abbiamo che $\exists \xi_i^{(2)} \in [\xi_i^{(1)}, \xi_{i+1}^{(1)}]$ t. c. $e''(\xi_i^{(2)}) = 0$ per $i = 0, \dots, r-2$ e così via. Alla fine si avrà che $\exists \xi_i^{(r)} \in [a, b]$ t. c. $e^{(r)}(\xi_i^{(r)}) = 0$. Quindi, se $0 = e^{(r)}(\xi) = f^{(r)}(\xi) - p^{(r)}(\xi)$, sapendo che $p^{(r)}(\xi) = r! f[x_0, \dots, x_r]$ si giunge alla tesi. Osserviamo che se $x_0 = x_1 = \dots = x_r$ si avrà $w_i(x) = \prod_{j=0}^{i-1} (x - x_j) = (x - x_0)^i$, quindi la differenza divisa sarà uguale al polinomio di Taylor di $f(x)$ calcolato nel punto x_0 .

- Dimostriamo la tesi usando la definizione di differenza divisa, otteniamo quindi:

$$\begin{aligned} & \frac{1}{x_{r+1} - x_0} \left[\sum_{j=1}^{r+1} \frac{f_j}{\prod_{k=1, k \neq j}^{r+1} (x_j - x_k)} - \sum_{j=0}^r \frac{f_j}{\prod_{k=0, k \neq j}^r (x_j - x_k)} \right] = \\ & \frac{1}{x_{r+1} - x_0} \left[\sum_{j=1}^r f_j \left(\frac{1}{\prod_{k=1, k \neq j}^{r+1} (x_j - x_k)} - \frac{1}{\prod_{k=0, k \neq j}^r (x_j - x_k)} \right) + \right. \\ & \quad \left. + \frac{f_{r+1}}{\prod_{k=1, k \neq r+1}^{r+1} (x_{r+1} - x_k)} - \frac{f_0}{\prod_{k=0, k \neq 0}^r (x_0 - x_k)} \right] = \\ & = \frac{f_{r+1}}{\prod_{k=0, k \neq r+1}^{r+1} (x_{r+1} - x_k)} + \frac{f_0}{\prod_{k=0, k \neq 0}^{r+1} (x_0 - x_k)} + \\ & + \frac{1}{x_{r+1} - x_0} \sum_{j=1}^r \frac{f_j}{\prod_{k=1, k \neq j}^r (x_k - x_j)} \left(\frac{1}{x_j - x_{r+1}} - \frac{1}{x_j - x_0} \right) \end{aligned}$$

Dove l'ultima componente vale

$$\sum_{j=1}^r \frac{f_j}{\prod_{k=0, k \neq j}^{r+1} (x_j - x_k)}$$

Pertanto la somma delle tre componenti risulta proprio $f[x_0, \dots, x_{r+1}]$

Esercizio 4.6

Costruire una *function* MATLAB che implementi in modo efficiente l'algoritmo 4.1.

Svolgimento

Codice MATLAB 4.6

Osservazioni: il ciclo `for` più interno, eseguito n volte, fa due sottrazioni ed una divisione, quindi il costo totale dell'algoritmo sarà dato da $3 \frac{n(n+1)}{2} \approx \frac{3}{2}n^2 \text{ flops}$.

Esercizio 4.7

Dimostrare che il seguente algoritmo, che riceve in ingresso i vettori x e f prodotti dalla *function* dell'esercizio precedente, valuta il corrispondente polinomio interpolante di Newton in punto xx assegnato.

```
p = f(n + 1)
for k = n:-1:1
    p = p*(xx - x(k)) + f(k)
end
```

Qual è il suo costo computazionale? Confrontarlo con quello dell'algoritmo precedente. Costruire, quindi, una corrispondente *function* MATLAB che lo implementi efficientemente (contemplare la possibilità che xx sia un vettore).

Svolgimento

Il costo computazionale dell'algoritmo generalizzato di Horner è di $3n \text{ flops}$, in quanto svolge per n volte 3 operazioni (una sottrazione, un'addizione ed una moltiplicazione). Quello per il calcolo della differenza divisa, come detto precedentemente, è di circa $\frac{3}{2}n^2 \text{ flops}$.

Codice MATLAB 4.7

Dato un vettore x di ascisse d'interpolazione di lunghezza $n + 1$ e un vettore xx di lunghezza m contenente l'insieme dei punti dove si vuole valutare il polinomio, il costo computazionale del codice sopra referenziato è pari a $3n * m \text{ flops}$. Difatti, nel ciclo interno vengono eseguite 3 operazioni: una moltiplicazione, una sottrazione ed un'addizione. Queste 3 operazioni vengono eseguite n volte, e tutto questo viene iterato per m volte.

Esercizio 4.8

Costruire una *function* MATLAB che implementi in modo efficiente l'algoritmo 4.2.

Svolgimento

Presentiamo il codice dell'algoritmo richiesto, ovvero il calcolo della differenza divisa di Hermite. L'implementazione segue semplicemente l'algoritmo 4.2 del libro di testo, dove i parametri in input sono le $2n + 2$ ascisse caratteristiche di Hermite ed il suo vettore tipico

$(f(x_0), f'(x_0), f(x_1), f'(x_1), \dots, f(x_n), f'(x_n))$. Restituisce in output il vettore delle differenze divise di Hermite.

[Codice MATLAB 4.8](#)

Esercizio 4.9

Si consideri la funzione $f(x) = (x - 1)^9$. Utilizzando le function degli esercizi 6-8, valutare i polinomi interpolanti di Newton e di Hermite sulle ascisse:

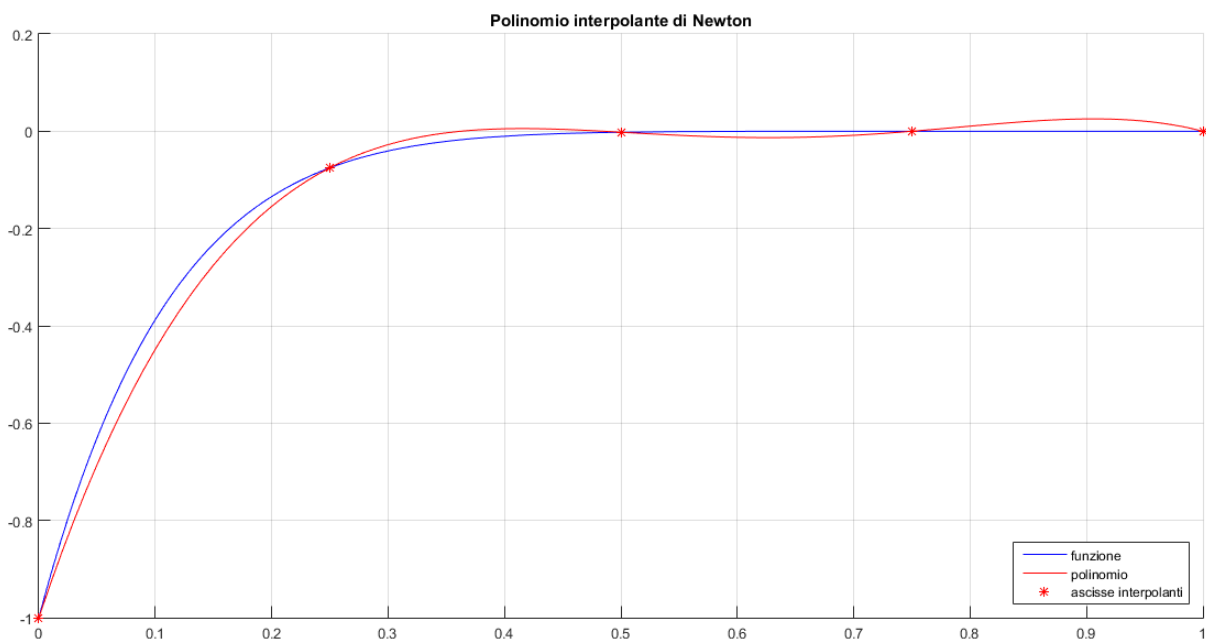
$$0, 0.25, 0.5, 0.75, 1$$

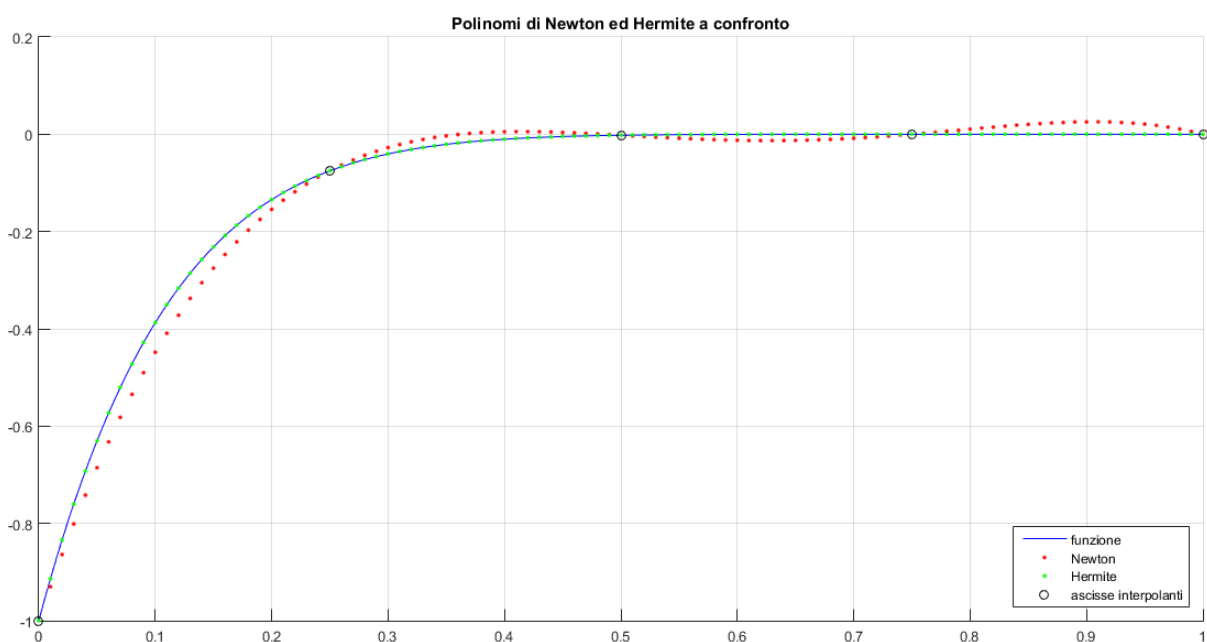
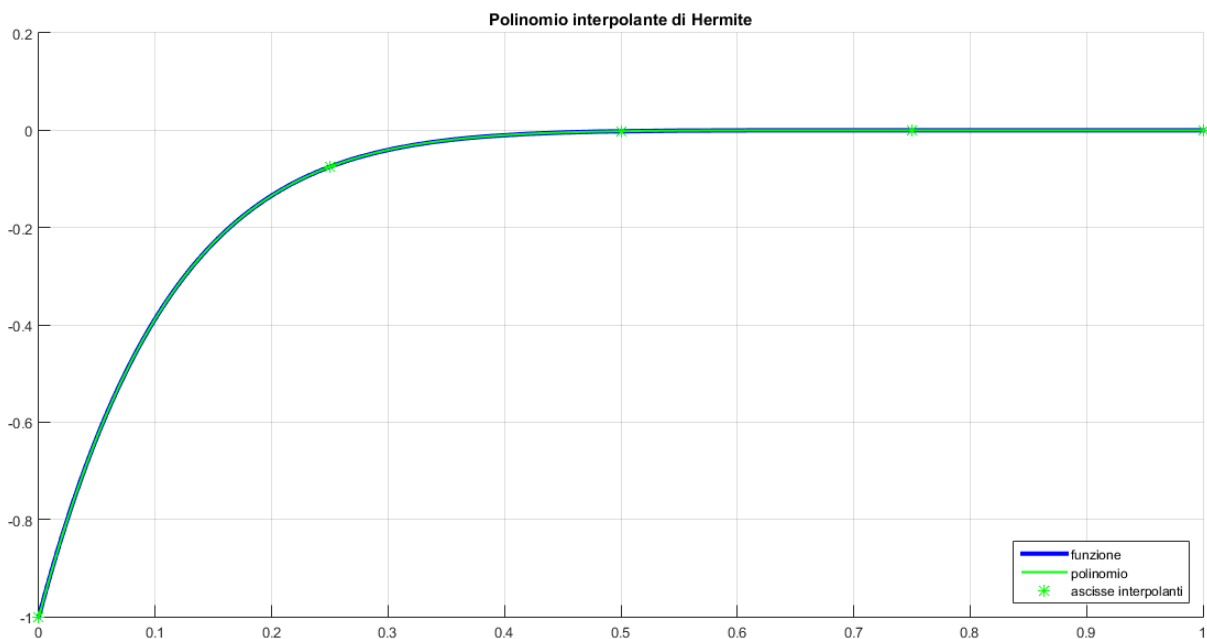
per $x = \text{linspace}(0,1,101)$. Raffigurare, quindi, (e spiegare) i risultati.

Svolgimento

Presentiamo lo script che esegue quanto richiesto, con grafici annessi. Lo script definisce inizialmente la funzione, le ascisse interpolanti ed i corrispettivi valori assunti dalla funzione. Calcola quindi la differenza divisa per Newton. Per calcolare quella di Hermite vengono usate alcune funzioni di librerie MATLAB (*reshape*, *matlabFunction*, *diff* e altre, oltre al calcolo simbolico) coadiuvanti la generazione dei parametri da dare in input alla *function* 4.8. Viene quindi generato il *linspace* richiesto ed usato l'algoritmo di Horner generalizzato (4.7) per valutare entrambi i polinomi. Viene infine prodotta la parte grafica.

[Codice MATLAB 4.9](#)





Dal momento che abbiamo solo 5 ascisse, il polinomio interpolante di Newton sarà di quarto grado, mentre con le 10 ascisse del polinomio interpolante di Hermite otterremo un polinomio di nono grado. Questo significa, come si evince dai grafici mostrati, che il polinomio interpolante di Newton non approssima bene la funzione quanto quello di Hermite, che è dello stesso grado della funzione polinomiale data e pertanto coincide con essa.

Esercizio 4.10

Quante ascisse di interpolazione equidistanti sono necessarie per approssimare la funzione $\sin(x)$ sull'intervallo $[0, 2\pi]$, con un errore di interpolazione inferiore a 10^{-6} ?

Svolgimento

Data l'espressione dell'errore:

$$e(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!} w_{n+1}(x) \quad \text{con } \xi \in [\min\{x_0, x\}, \max\{x_n, x\}]$$

vogliamo trovare una maggiorazione dei termini che lo compongono tale che il risultato sia minore di 10^{-6} . Data la derivabilità della funzione in esame, e considerato l'intervallo, avremo che:

$$f^{(n+1)}(\xi) \leq 1 \quad \text{con } \xi \in [\min\{0, x\}, \max\{2\pi, x\}]$$

Per trovare una maggiorazione adeguata del termine $w_{n+1}(x)$ notiamo che:

$$w_{n+1}(x) = \prod_{j=0}^n (x - x_j)$$

Dato l'intervallo considerato, possiamo maggiorare il termine $|x - x_j|$ con 2π . Pertanto avremo:

$$\prod_{j=0}^n (x - x_j) \leq \prod_{j=0}^n 2\pi = (2\pi)^{n+1} \quad \forall j$$

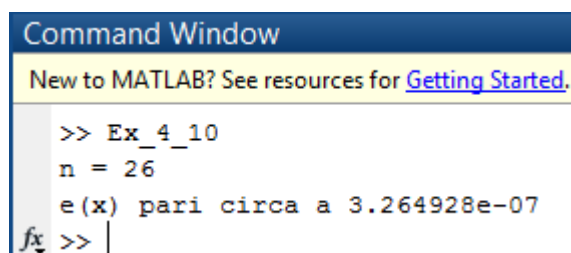
Riscriviamo infine l'espressione dell'errore derivante:

$$e(x) \leq \frac{(2\pi)^{n+1}}{(n+1)!}$$

Si tratta ora semplicemente di trovare una n tale che l'errore sia minore di 10^{-6} . Compito facilmente svolto dal seguente codice MATLAB:

```
n = 1;
e = 1;
while e > 10^-6
    n = n + 1;
    e = ( (2*pi)^(n + 1) ) / factorial(n + 1);
end
fprintf('n = %d \ne(x) pari circa a %d\n', n, e);
```

Stampa sulla finestra di comando del risultato:



```
Command Window
New to MATLAB? See resources for Getting Started.
>> Ex_4_10
n = 26
e(x) pari circa a 3.264928e-07
fx >> |
```

Esercizio 4.11

Verificare sperimentalmente che, considerando le ascisse d'interpolazione equidistanti

$$x_i = a + i * h, \quad i = 0, 1, \dots, n, \quad h = \frac{b - a}{n}$$

su cui si definisce il polinomio $p(x)$ interpolante $f(x)$, l'errore $\|f - p\|$ diverge, al crescere di n , nei seguenti tre casi:

1. esempio di Runge:

$$f(x) = \frac{1}{1 + x^2} \quad [a, b] \equiv [-5, 5]$$

2. esempio di Bernstein:

$$f(x) = |x| \quad [a, b] \equiv [-1, 1]$$

3. esempio del pdf di complemento:

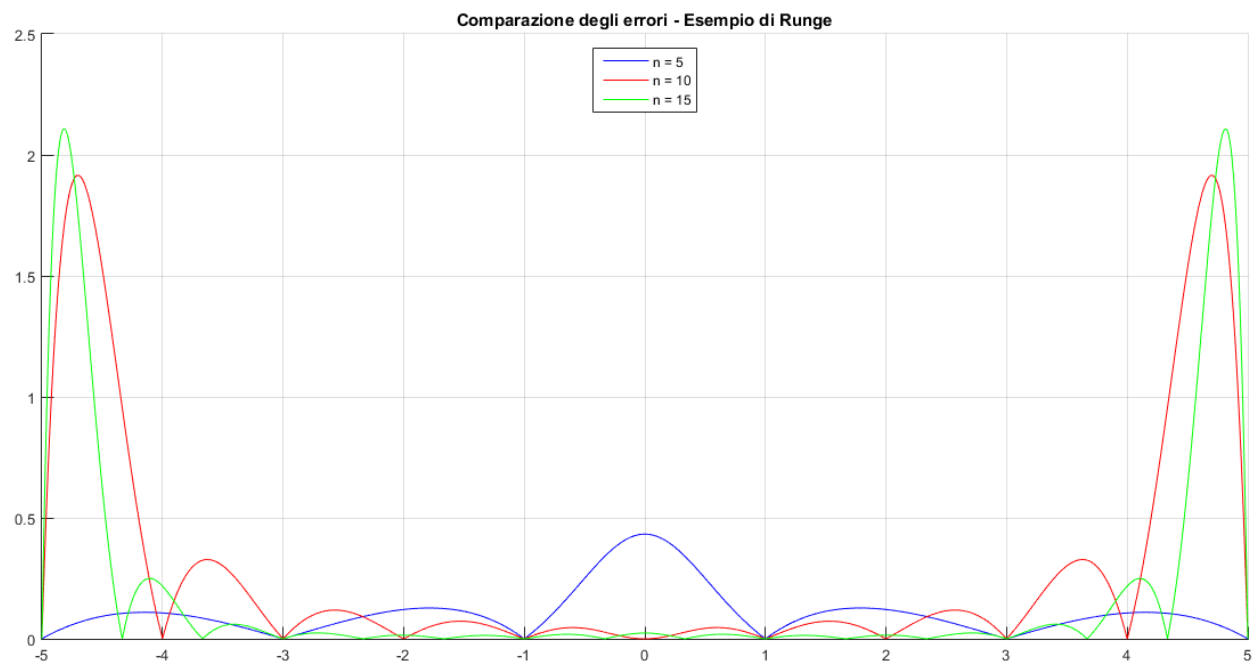
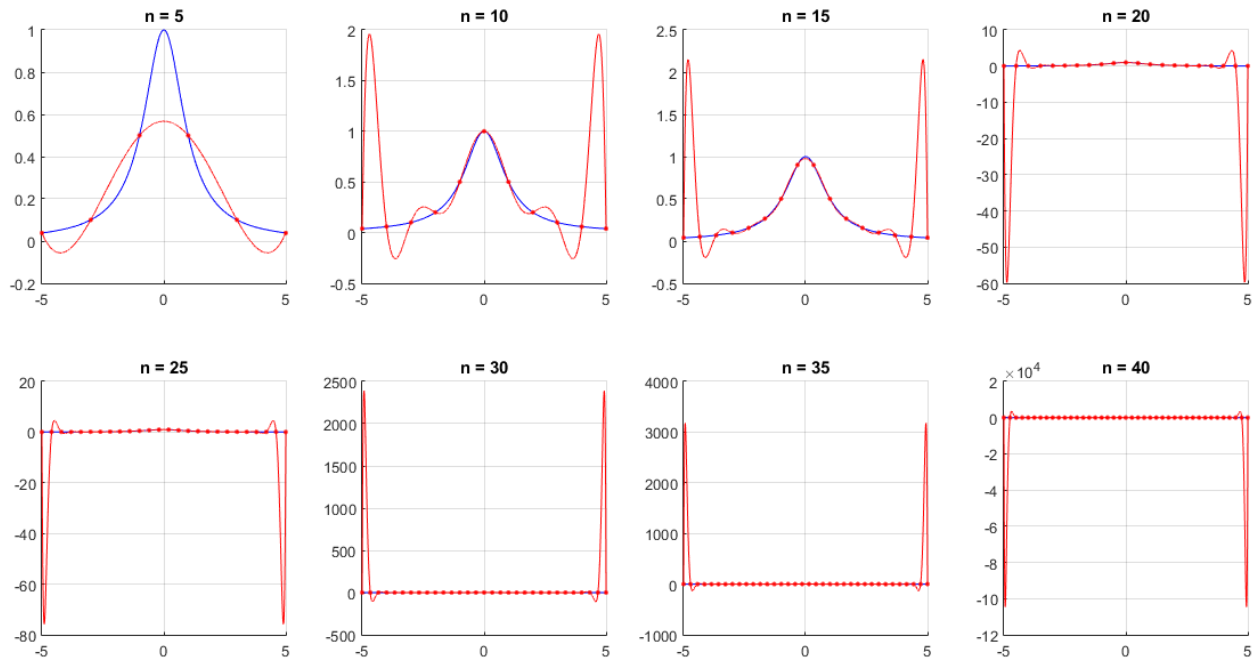
$$f(x) = \frac{1}{(x^4 - x^2 + 1)} \quad [a, b] \equiv [-5, 5]$$

Svolgimento

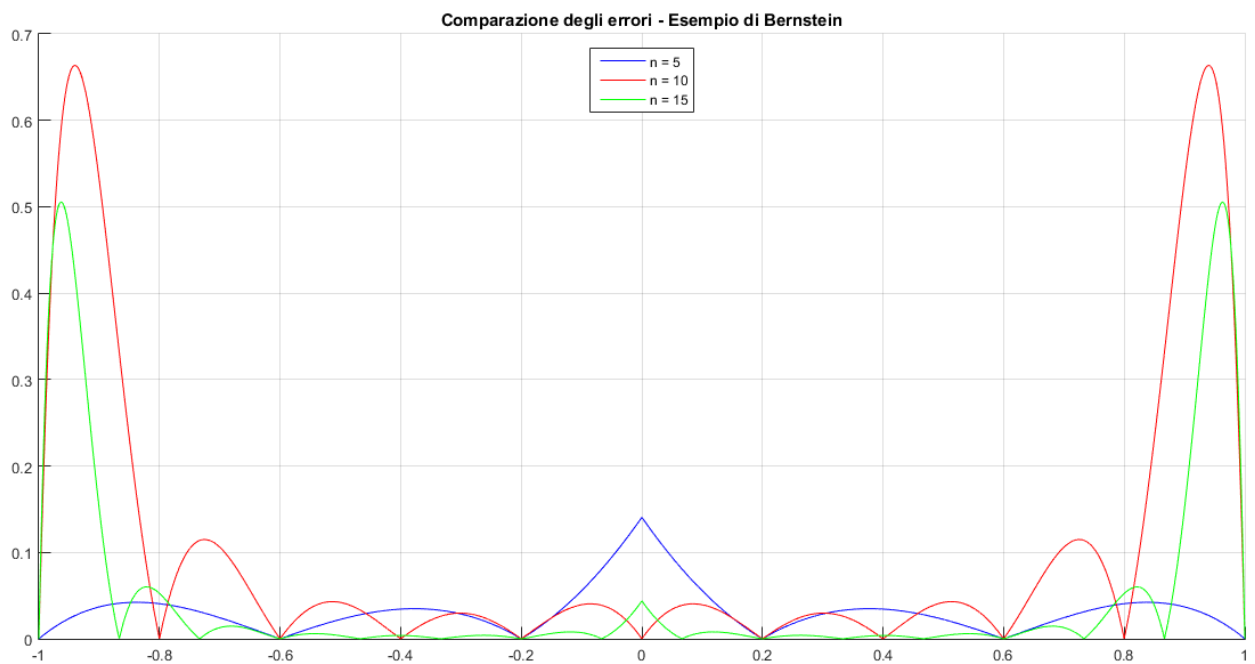
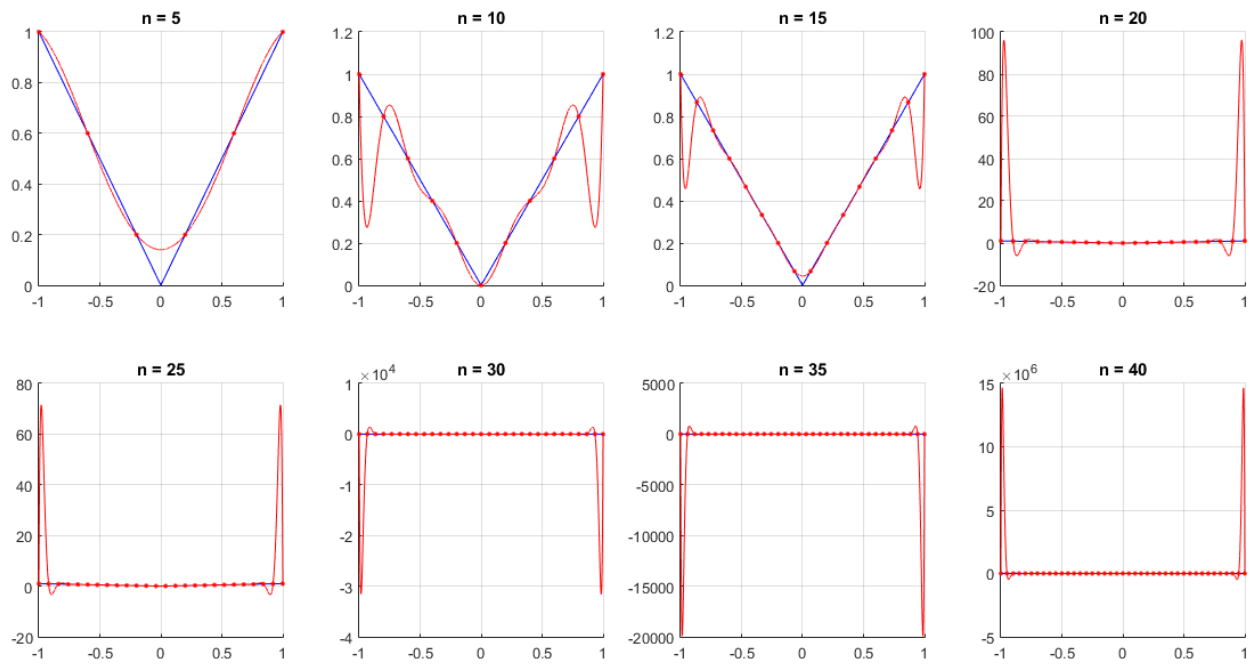
Presentiamo due codici: uno script ed una *function*. Lo script inizializza le funzioni della traccia ed organizza l'output della *function* che richiama iterativamente per ogni funzione, testando ogni funzione su ciascun intervallo scelto. La *function* prende in input la funzione, i suoi estremi, un valore n corrispondente al numero di intervalli testati, un booleano per scegliere se calcolare le ascisse di Chebyshev (esercizio 4.15) o usare una partizione uniforme (come richiesto dalla traccia di questo esercizio) ed un altro booleano per abilitare o meno la parte grafica. Restituisce il vettore delle valutazioni del polinomio interpolante e l'errore $\|f - p\|$. Per valutare il polinomio, la *function* richiama prima il metodo per calcolare le differenze divise e poi quello dell'algoritmo di Horner generalizzato. Si occupa inoltre della costruzione della parte grafica. Lo script conclude l'esecuzione del tutto stampando sulla finestra di comando una tabella riportante, per ciascun intervallo scelto, l'errore delle tre funzioni prese in esame

Presentiamo, per ciascuna funzione, un grafico relativo al comportamento del corrispondente polinomio interpolante su $n + 1$ ascisse equispaziate negli intervalli dati per $n = 5, 10, 15, 20, 25, 30, 35, 40$ ed un grafico che mette in relazione gli errori per $n = 5, 10, 15$.

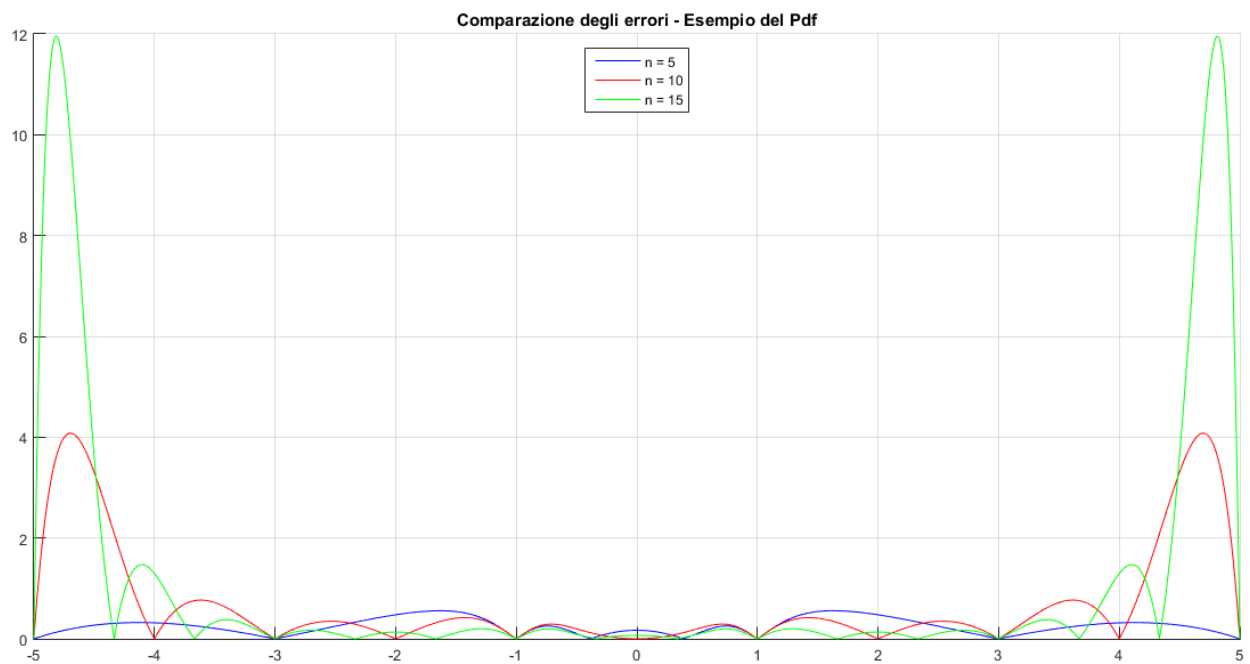
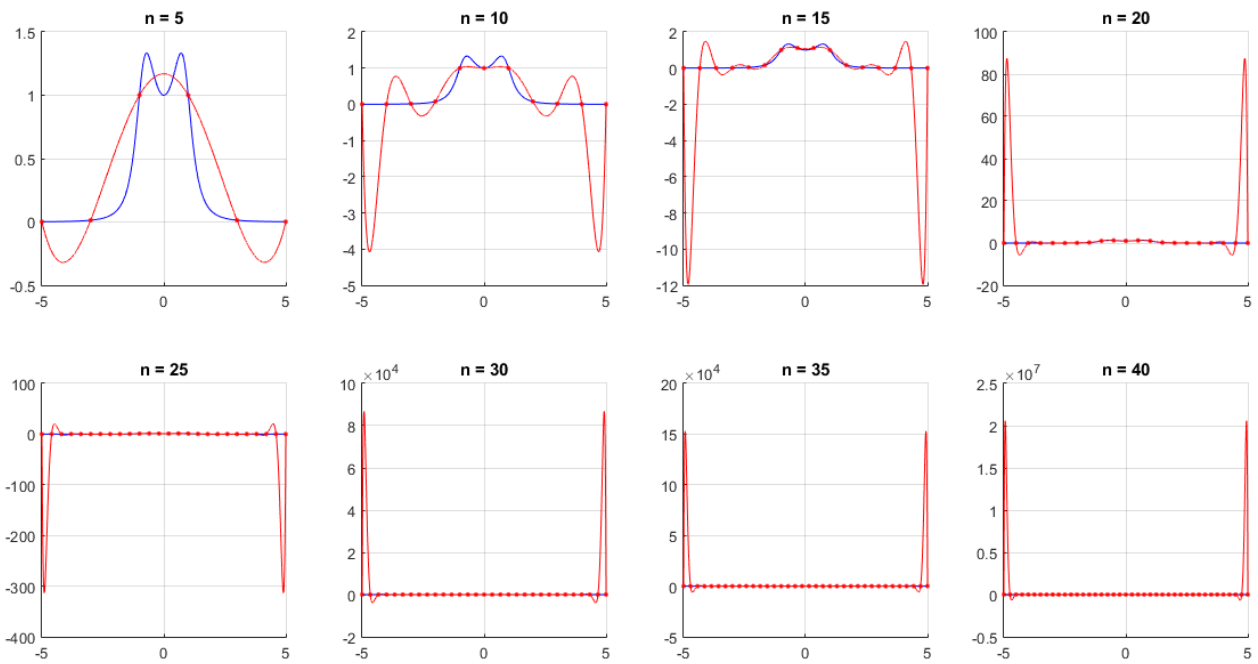
Esempio di Runge:



Esempio di Bernstein:



Esempio del pdf di complemento:



Presentiamo infine un'estratto della tabella degli errori generata dallo script. Usiamo un linspace di 10001 punti per $n = 10, 20, 30, 40$.

n	<i>Runge</i>	<i>Bernstein</i>	<i>Pdf di complemento</i>
10	≈ 1.916	≈ 0.664	≈ 4.079
20	≈ 59.822	≈ 95.189	≈ 87.487
30	≈ 2388.281	≈ 31581.816	≈ 86847.134
40	≈ 104667.687	≈ 14656781.905	≈ 20578546.478

Notiamo come, scegliendo ascisse equispaziate, all'aumentare di n aumenti l'errore.

[Codice MATLAB 4.11](#) (la *function*)

[Codice MATLAB 4.11Script](#)

Esercizio 4.12

Dimostrare che, se $x \in [-1, 1]$, allora:

$$\tilde{x} \equiv \frac{a+b}{2} + \frac{b-a}{2}x \in [a, b]$$

Viceversa, se $\tilde{x} \in [a, b]$, allora:

$$x \equiv \frac{2\tilde{x} - a - b}{b - a} \in [-1, 1]$$

Concludere che è sempre possibile trasformare il problema di interpolazione in uno definito sull'intervallo $[-1, 1]$, e viceversa.

Svolgimento

Dimostrazione:

con $x \in [-1, 1]$ distinguiamo due casi:

$$\begin{aligned} x = -1 &\rightarrow \tilde{x} = \frac{a+b}{2} - \frac{b-a}{2} = \frac{2a}{2} = a \\ x = 1 &\rightarrow \tilde{x} = \frac{a+b}{2} + \frac{b-a}{2} = \frac{2b}{2} = b \end{aligned}$$

con $\tilde{x} \in [a, b]$ altrettanto:

$$\begin{aligned} \tilde{x} = a &\rightarrow x = \frac{2a - a - b}{b - a} = \frac{a - b}{b - a} = -1 \\ \tilde{x} = b &\rightarrow x = \frac{2b - a - b}{b - a} = \frac{b - a}{b - a} = 1 \end{aligned}$$

Pertanto è sempre possibile traslare opportunamente l'intervallo dato.

Esercizio 4.13

Dimostrare le proprietà dei polinomi di Chebyshev di prima specie elencate nel Teorema 4.9.

Svolgimento

Definita la famiglia dei polinomi di Chebyshev di prima specie:

$$T_0(x) \equiv 1$$

$$T_1(x) = x$$

$$T_{k+1}(x) = 2xT_k(x) - T_{k-1}(x) \quad k = 1, 2, \dots$$

abbiamo le seguenti proprietà:

- $T_k(x)$ è un polinomio di grado esatto k
- Il coefficiente principale di $T_k(x)$ è 2^{k-1} , $k = 1, 2, \dots$
- La famiglia di polinomi $\{\hat{T}_k\}$, in cui $\hat{T}_0(x) = T_0(x)$, $\hat{T}_k(x) = 2^{1-k} T_k(x)$ con $k = 1, 2, \dots$ è una famiglia di polinomi monici di grado k , $k = 0, 1, \dots$
- Ponendo $x = \cos(\theta)$, $\theta \in [0, \pi]$ si ottiene $T_k(x) \equiv T_k(\cos\theta) = \cos(k\theta)$, $k = 0, 1, \dots$

Dimostrazioni:

- Con $k = 0$ si ha $T_0(x) = 1$, ovvero il polinomio di grado 0, per il caso in cui $k > 0$ usando l'ipotesi induttiva otteniamo $2xT_k(x) - T_{k-1}(x)$ dove $T_k(x)$ è un polinomio di grado esatto k , quindi $T_{k+1}(x)$ è un polinomio di grado $k + 1$.
- Con $k = 1$ si ha che per $T_1(x)$ il coefficiente è $2^{1-1} = 2^0 = 1$, per il caso in cui $k > 1$, abbiamo che $T_{k+1}(x) = 2xT_k(x) - T_{k-1}(x)$; dove per ipotesi induttiva concludiamo che 2^{k-1} è il coefficiente principale di $T_k(x)$, quindi quello di $T_{k+1}(x)$ è $2 * 2^{k-1} = 2^k$
- Per i polinomi di grado k si ha che $\hat{T}_k(x)$ coincide con il grado di $T_k(x)$ che è appunto di grado k . Per il coefficiente principale di $\hat{T}_k(x)$ si ha $2^{1-k} 2^{k-1} = 1$ quindi il polinomio risulta monico come ci aspettavamo.
- Con $k = 0$ si ha $T_0(\cos(\theta)) = \cos(0\theta) = 1 = T_0(x)$
Con $k = 1$ si ha $T_1(\cos(\theta)) = \cos(\theta) = x = T_1(x)$
Per $k > 1$ si ha per ipotesi d'induzione $T_{k+1}(\cos(\theta)) = 2\cos(\theta)T_k(\cos(\theta)) - T_{k-1}(\cos(\theta))$, dato che $T_k(\cos(\theta)) = \cos(k\theta)$ e $T_{k-1}(\cos(\theta)) = \cos((k-1)\theta)$ si ottiene
 $T_{k+1}(\cos(\theta)) = 2\cos(\theta)\cos(k\theta) - \cos((k-1)\theta) = \cos(k+1)\theta + \cos(k-1)\theta - \cos(k-1)\theta = \cos(k+1)\theta = T_{k+1}(x)$

Esercizio 4.14

Quali diventano le ascisse di Chebyshev, per un problema definito su un generico intervallo $[a, b]$?

Svolgimento

La ascisse di Chebyshev nell'intervallo $[-1,1]$ sono così definite:

$$x_{n-i} = \cos\left(\frac{(2i+1)\pi}{2(n+1)}\right) \quad i = 0, \dots, n$$

Come dimostrato nell'esercizio 12, possiamo traslare tali ascisse su un intervallo generico $[a, b]$ ottenendo le generiche ascisse di Chebyshev:

$$\tilde{x}_{n-i} \equiv \frac{a+b}{2} + \frac{b-a}{2} \cos\left(\frac{(2i+1)\pi}{2(n+1)}\right) \in [a, b]$$

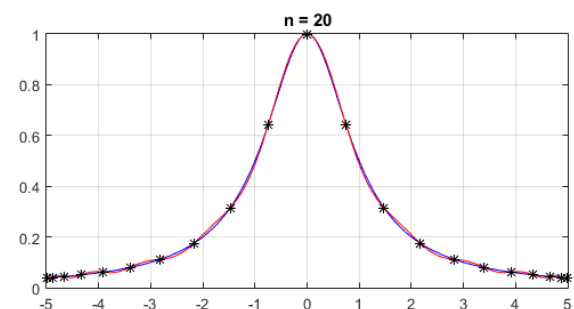
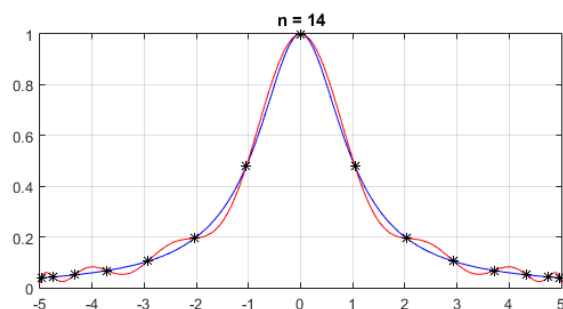
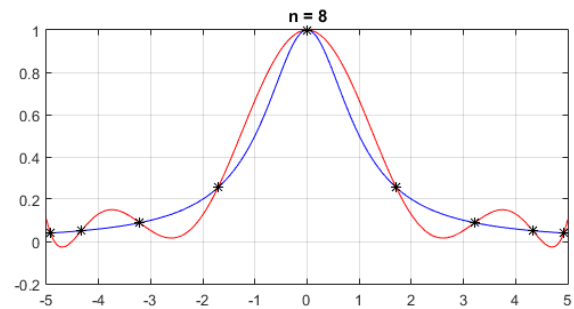
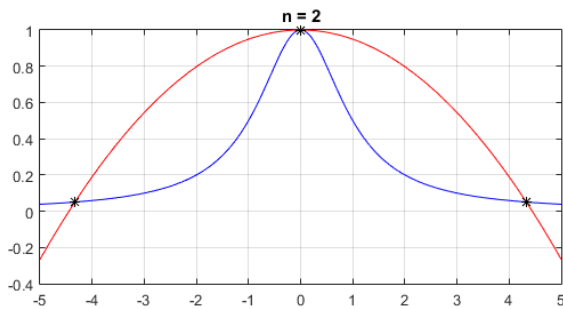
Esercizio 4.15

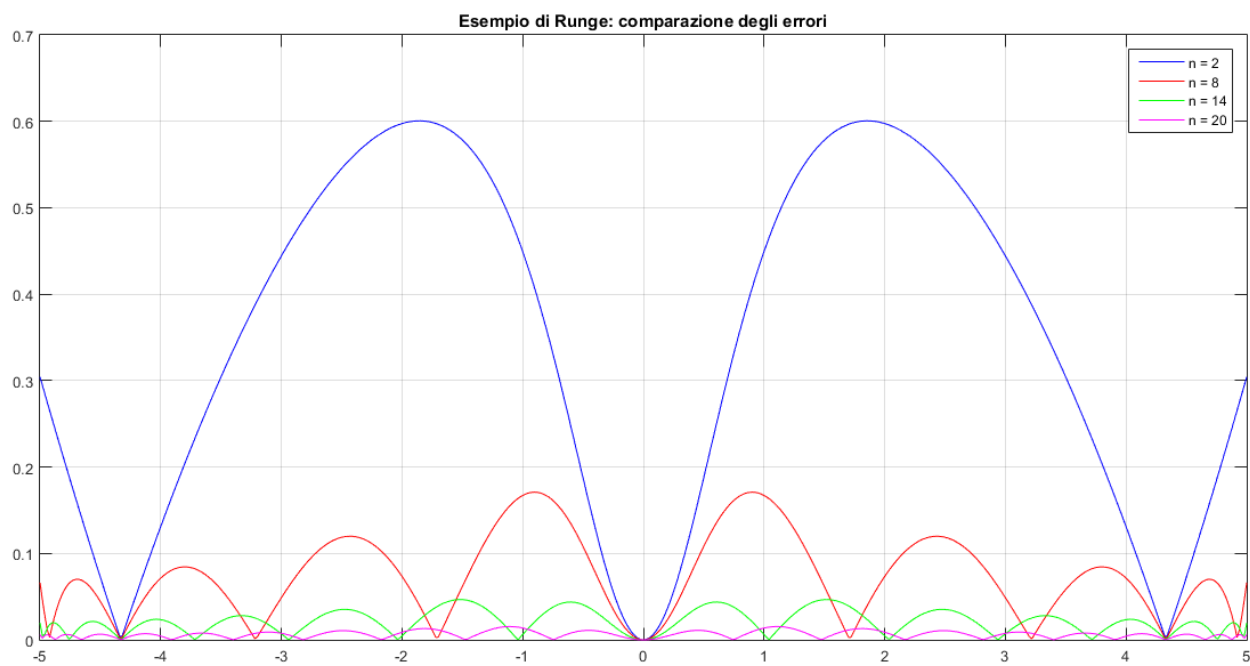
Utilizzare le ascisse di Chebyshev per approssimare gli esempi visti nell'esercizio 11, per $n = 2, 4, 6, \dots, 40$

Svolgimento

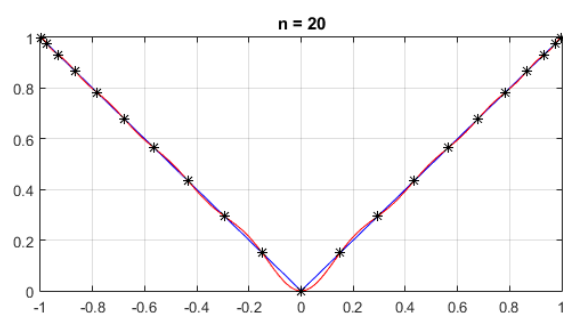
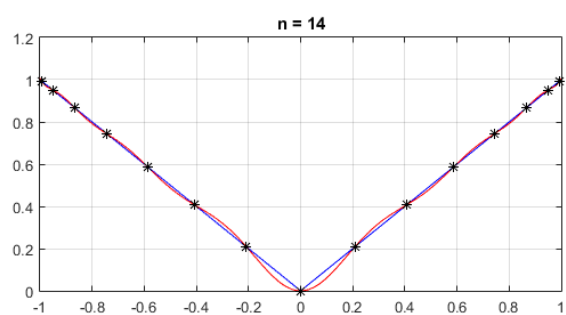
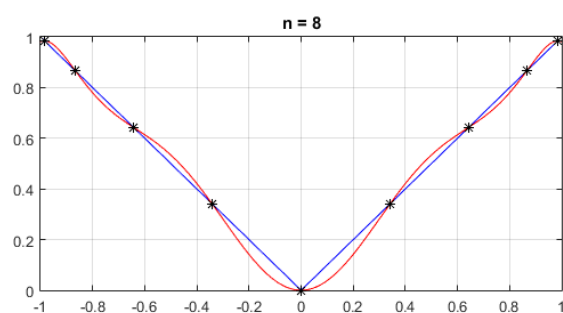
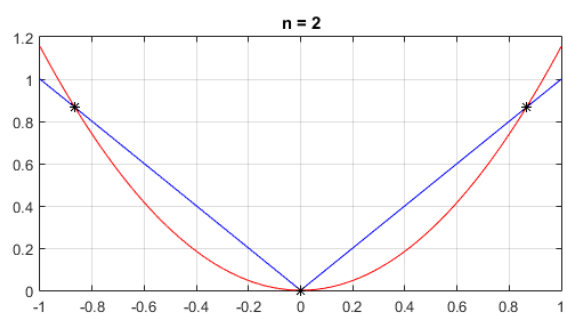
Mostriamo, analogamente a quanto fatto per l'esercizio 4.11, i vari grafici delle funzioni, considerando sempre $n+1$ ascisse interpolanti e per i seguenti valori di $n = 2, 8, 14, 20$.

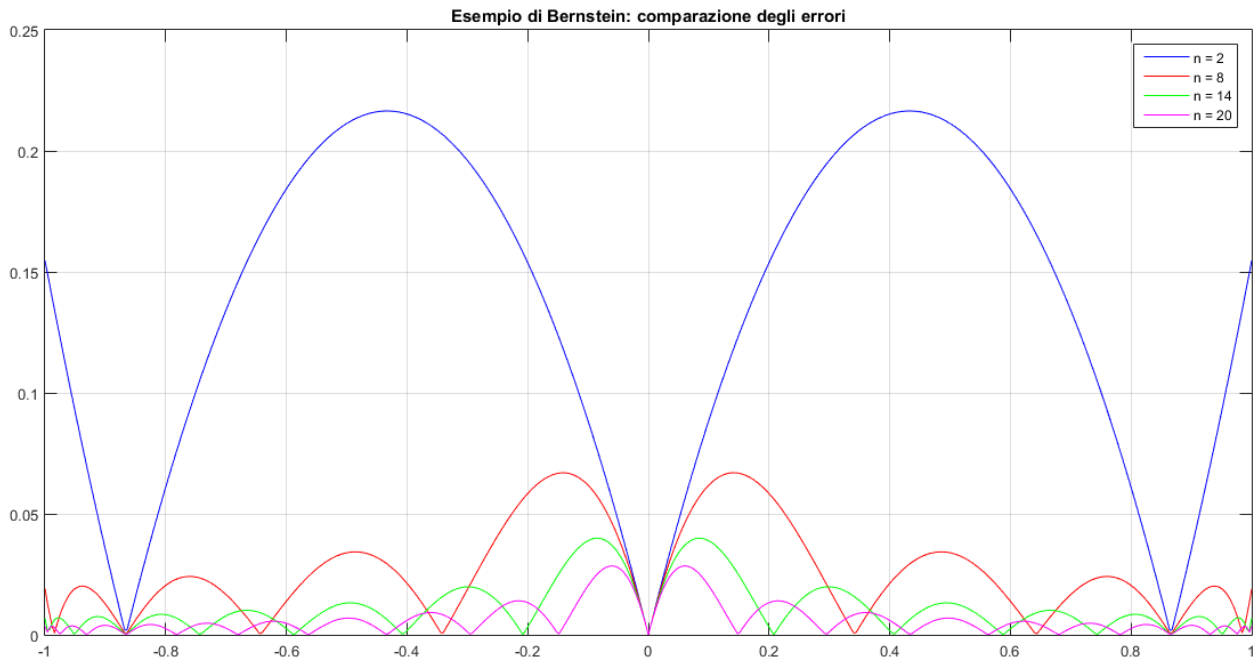
Esempio di Runge:



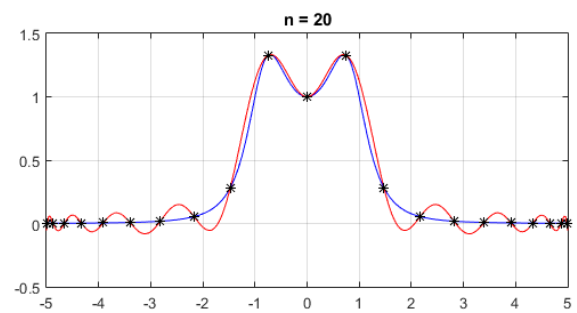
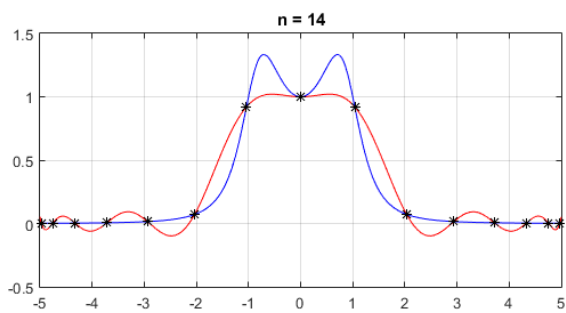
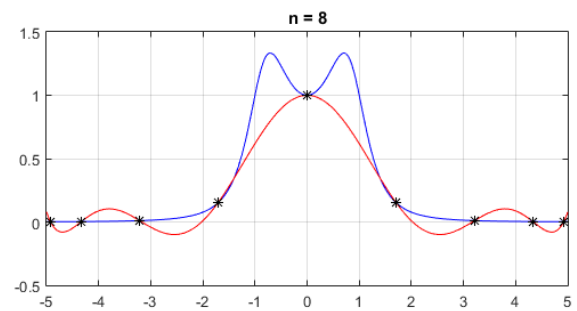
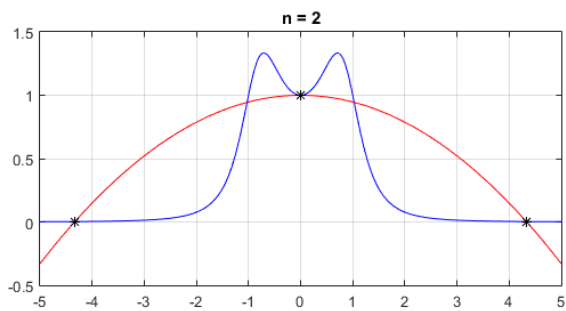


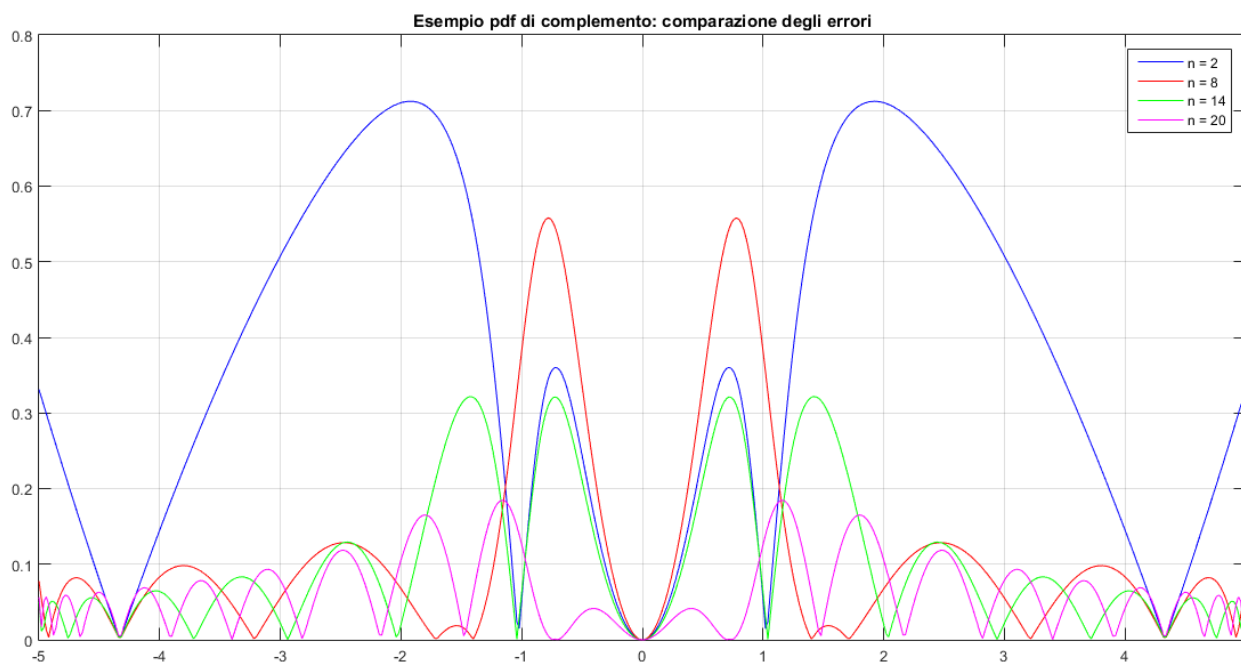
Esempio di Bernstein:





Esempio del pdf di complemento:





Riportiamo infine la tabella relativa agli errori, con $n = 2, 4, 6, \dots, 40$

n	<i>Runge</i>	<i>Bernstein</i>	<i>Pdf di complemento</i>
2	≈ 0.6	≈ 0.2	≈ 0.7
4	≈ 0.4	≈ 0.1	≈ 0.43
6	≈ 0.26	$\approx 8.6 * 10^{-2}$	≈ 0.48
8	≈ 0.17	$\approx 6.6 * 10^{-2}$	≈ 0.55
10	≈ 0.109	$\approx 5.4 * 10^{-2}$	≈ 0.57
12	$\approx 6.9 * 10^{-2}$	$\approx 4.6 * 10^{-2}$	≈ 0.48
14	$\approx 4.6 * 10^{-2}$	$\approx 3.9 * 10^{-2}$	≈ 0.32
\vdots	\vdots	\vdots	\vdots
38	$\approx 4.3 * 10^{-4}$	$\approx 1.5 * 10^{-2}$	$\approx 3.2 * 10^{-2}$
40	$\approx 2.8 * 10^{-4}$	$\approx 1.4 * 10^{-2}$	$\approx 1.9 * 10^{-2}$

Coerentemente con quanto visto a lezione, usando le ascisse di Chebyshev, all'aumentare di n si abbate l'errore. Vale la pena ricordare che, per come sono costruiti i polinomi di Chebyshev di prima specie, si ha $\|w_{n+1}\| = 2^{-n}$, pertanto l'espressione dell'errore relativa a tali polinomi diventa $\|e\| = \frac{\|f^{(n+1)}\|}{(n+1)!2^n}$.

Per quanto riguarda la parte implementativa, vedere il codice presentato nello svolgimento dell'esercizio 4.11.

Esercizio 4.16

Verificare che la fattorizzazione LU della matrice dei coefficienti del sistema tridiagonale delle spline naturali è data da:

$$L = \begin{pmatrix} 1 & & & \\ l_2 & 1 & & \\ & \ddots & \ddots & \\ & & l_{n-1} & 1 \end{pmatrix}$$

$$U = \begin{pmatrix} u_1 & \xi_1 & & \\ & u_2 & \ddots & \\ & & \ddots & \xi_{n-2} \\ & & & u_{n-1} \end{pmatrix}$$

con $u_1 = 2, l_i = \frac{\varphi_i}{u_{i-1}}, u_i = 2 - l_i \xi_{i-1}, i = 2, \dots, n-1$.

Scrivere una *function* MATLAB che implementi efficientemente la risoluzione del sistema lineare tridiagonale.

Svolgimento

Questa è la matrice dei coefficienti:

$$A = \begin{pmatrix} 2 & \xi_1 & & & \\ \varphi_2 & 2 & \xi_2 & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & \xi_{n-2} \\ & & & \varphi_{n-1} & 2 \end{pmatrix}$$

Osserviamo che tale matrice è diagonale dominante per righe dato che $\varphi_i = \frac{h_i}{h_i + h_{i+1}}$ e $\xi_i = \frac{h_{i+1}}{h_i + h_{i+1}}$ con $i = 1, \dots, n-1$, e quindi $\varphi_i \xi_i > 0$ e $\varphi_i + \xi_i = 1$ con $i = 1, \dots, n-1$, pertanto la matrice è fattorizzabile LU . Se moltiplichiamo L con U otteniamo i coefficienti della matrice A . A verifica di questo fatto:

1. $a_{1,1} = 1 * u_1 = 2$
2. $a_{1,2} = 1 * \xi_1 + 0 * u_2 = \xi_1$
3. $a_{i,i-1} = 0 * \xi_{i-2} + l_i u_{i-1} + 1 * 0 = l_i u_{i-1} = \frac{\varphi_i}{u_{i-1}} u_{i-1} = \varphi_i$, per $i > 1$
4. $a_{i,i} = l_i \xi_{i-1} + 1 * u_i = l_i \xi_{i-1} + 2 - l_i \xi_{i-1} = 2$, per $i > 1$
5. $a_{i,i+1} = l_i * 0 + 1 * \xi_i + 0 * u_{i+1} = \xi_i$, per $i > 1$

Andiamo quindi a risolvere il sistema $A \underline{m} = \underline{d}$. Per far questo bisogna risolvere il sistema:

$$\begin{cases} L \underline{y} = \underline{d} \\ U \underline{m} = \underline{y} \end{cases}$$

Risolvendo il $\underline{L}\underline{y} = 6\underline{d}$ otteniamo:

$$y_1 = 6d_1$$

$$y_i = 6d_i - l_i y_{i-1} \text{ con } i = 2, \dots, n-1$$

Risolvendo $\underline{U}\underline{m} = \underline{y}$ otteniamo:

$$m_{n-1} = \frac{y_{n-1}}{u_{n-1}}$$

$$m_i = \frac{y_i - \xi_i x_{i+1}}{u_i} \text{ con } i = n-2, \dots, 1$$

Siamo ora in grado di scrivere la relativa funzione in MATLAB:

[Codice MATLAB 4.16.1](#)

Tale implementazione non fa uso dell'algoritmo di fattorizzazione LU , bensì sfrutta le ottimizzazioni possibili appena calcolate. Inoltre, per il calcolo del vettore contenente le differenze divise, abbiamo scritto quest'altra piccola funzione:

[Codice MATLAB 4.16.2](#)

Esercizio 4.17

Generalizzare la fattorizzazione del precedente esercizio 16 al caso della matrice dei coefficienti del sistema lineare (4.59). Scrivere una corrispondente *function* MATLAB che risolva efficacemente questo sistema.

Svolgimento

La matrice dei coefficienti in questione è la seguente:

$$A = \begin{pmatrix} 1 & 0 & & & & & & \\ \varphi_1 & 2 - \varphi_1 & \xi_1 - \varphi_1 & & & & & \\ & \varphi_2 & 2 & \xi_2 & & & & \\ & & \ddots & \ddots & \ddots & & & \\ & & & \varphi_{n-2} & 2 & \xi_{n-2} & & \\ & & & & \varphi_{n-1} - \xi_{n-1} & 2 - \xi_{n-1} & \xi_{n-1} & \\ & & & & & 0 & 1 & \end{pmatrix}$$

Analogamente a quanto visto nell'esercizio precedente, troviamo i componenti delle matrici LU che fattorizzano A .

Per $i = 1$ abbiamo:

$$a_{1,1} = 1 = u_1 \rightarrow u_1 = 1$$

$$a_{1,2} = 0 = w_1 \rightarrow w_1 = 0$$

Per $i = 2$ abbiamo:

$$a_{2,1} = \varphi_1 = l_2 \rightarrow l_2 = \varphi_1$$

$$a_{2,2} = 2 - \varphi_1 = u_2 \rightarrow u_2 = 2 - \varphi_1$$

$$a_{2,3} = \xi_1 - \varphi_1 = w_2 \rightarrow w_2 = \xi_1 - \varphi_1$$

Per $i = 3$ abbiamo:

$$a_{3,2} = \varphi_2 = l_3 u_2 \rightarrow l_3 = \frac{\varphi_2}{u_2}$$

$$a_{3,3} = 2 = l_3 w_2 + u_3 \rightarrow u_3 = 2 - l_3 w_2$$

$$a_{3,4} = \xi_2 = w_3 \rightarrow w_3 = \xi_2$$

Per $i = 4, \dots, n-1$ abbiamo:

$$a_{i,i-1} = \varphi_{i-1} = l_i u_i \rightarrow l_i = \frac{\varphi_{i-1}}{u_{i-1}}$$

$$a_{i,i} = 2 = l_i w_{i-1} + u_i \rightarrow u_i = 2 - l_i w_{i-1}$$

$$a_{i,i+1} = \xi_{i-1} = w_i \rightarrow w_i = \xi_{i-1}$$

Per $i = n$ abbiamo:

$$a_{n,n-1} = \varphi_{n-1} - \xi_{n-1} = l_n u_{n-1} \rightarrow l_n = \frac{\varphi_{n-1} - \xi_{n-1}}{u_{n-1}}$$

$$a_{n,n} = 2 - \xi_{n-1} = l_n w_{n-1} + u_n \rightarrow u_n = 2 - \xi_{n-1} - l_n w_{n-1}$$

$$a_{n,n+1} = \xi_{n-1} = w_n \rightarrow w_n = \xi_{n-1}$$

Per $i = n+1$ abbiamo:

$$a_{n+1,n} = 0 = l_{n+1} u_n \rightarrow l_{n+1} = 0$$

$$a_{n+1,n+1} = 1 = l_{n+1} w_n + u_{n+1} \rightarrow u_{n+1} = 1 - l_{n+1} w_n = 1$$

Siamo pronti per risolvere il sistema $A\mathbf{m} = \mathbf{d}$. Risolviamo quindi:

$$\begin{cases} L\mathbf{y} = 6\mathbf{d} \\ U\mathbf{m} = \mathbf{y} \end{cases}$$

Svolgendo il sistema avremo:

$$y_1 = 6d_1$$

$$y_i = 6d_i - l_i y_{i-1} \quad \text{con } i = 2, \dots, n+1$$

continuando:

$$m_{n+1} = \frac{y_{n+1}}{u_{n+1}}$$

$$m_i = \frac{y_i - w_i m_{i+1}}{u_i} \quad \text{con } i = 1, \dots, n$$

Il calcolo delle soluzioni \hat{m}_i sarà quindi:

$$\hat{m}_1 = m_1 - m_2 - m_3$$

$$\hat{m}_i = m_i \quad \text{per } i = 2, \dots, n$$

$$\hat{m}_{n+1} = m_{n+1} - m_n - m_{n-1}$$

Presentiamo il codice MATLAB relativo:

[Codice MATLAB 4.17](#)

È stato implementato anche un controllo sul numero delle ascisse interpolanti la funzione: se minore di 4 questo significa che, date le condizioni da imporre per la costruzione di una spline di tipo *knot a not*, essa coincide con la funzione stessa.

Esercizio 4.18

Scrivere una *function* MATLAB che, noti gli $\{m_i\}$ in (4.47), determini l'espressione, polinomiale a tratti, della spline cubica (4.52) – (4.54).

Svolgimento

La *function* prende in input un vettore di ascisse interpolanti, relative valutazioni di funzione ed il vettore delle m proveniente da uno dei precedenti esercizi. Restituisce un vettore di n polinomi costituenti la tipica espressione della spline, presente sul libro di testo, usando il calcolo simbolico.

[Codice MATLAB 4.18](#)

Esercizio 4.19

Costruire una *function* MATLAB che implementi le spline cubiche naturali e quelle definite dalle condizioni not-a-knot.

Svolgimento

L'algoritmo prende in input un vettore di ascisse interpolanti, relative valutazioni di funzione ed un booleano che indica se la spline è di tipo *naturale* o *not a knot*. Costruisce i vettori φ e ξ atti alla fattorizzazione della matrice, richiama il metodo per il calcolo della differenza divisa per le spline e, a seconda del booleano, richiama il metodo scelto per il calcolo dei fattori m_i . Dopodiché non fa altro che chiamare il 4.18 e restituire quel che esso restituisce.

[Codice MATLAB 4.19](#)

Esercizio 4.20

Utilizzare la *function* dell'esercizio 4.19 per approssimare, su partizioni (4.41) uniformi con $n = 10, 20, 30, 40$, gli esempi proposti nell'esercizio 4.11.

Svolgimento

Nello svolgere questo esercizio abbiamo voluto esplorare alcune funzionalità di MATLAB. Presentiamo 4 codici: lo script *Ex_4_20_Script* all'interno dei suoi cicli richiama iterativamente la procedura *Ex_4_20_Grafica*(f, a, b, n, nak) dove f è la funzione presa in esame, a e b sono gli estremi della funzione, n è la n richiesta e nak è un booleano che indica se eseguire la costruzione di una spline *not a knot* o *naturale*. Al suo interno, tale procedura crea la partizione contenente le ascisse interpolanti, il vettore di valori assunti dalla funzione in corrispondenza della partizione ed il linspace sul quale valutare la spline. Richiama quindi le *function* *Ex_4_19*($x, f, type$) e *Ex_4_20*($p, s, xx, check$) per ottenere le valutazioni dei vari polinomi costituenti la spline su tutti i punti del linspace. Dopodiché disegna il grafico mettendo a confronto la funzione e la spline. Lo script organizza i vari grafici e conclude richiamando un altro script, *4.20_ErrTab*, dove viene disegnata una tabella `uitable` dai numerosi attributi che, per ciascuna n , in corrispondenza di ogni funzione, mostra una stima dell'errore commesso dalla nostra implementazione *not a knot* rispetto ai valori ottenuti valutando la spline con la libreria di MATLAB, che utilizza la stessa condizione. Viene mostrato anche il tempo impiegato per ciascuna valutazione.

La *function* *Ex_4_20*($p, s, xx, check$) è l'algoritmo vero e proprio di valutazione della spline: prende in argomento una partizione p , l'insieme s di polinomi costituenti la spline derivanti dalla funzione dell'esercizio precedente, un linspace xx sul quale valutare i polinomi ed un booleano $check$ che rende facoltativi i controlli sulla correttezza dei parametri in input. Difatti, tale metodo controlla anzitutto se la lunghezza della partizione è uguale a quella della partizione priva di eventuali doppi, eventualmente aggiustandola, e successivamente controlla se contiene almeno 3 ascisse interpolanti. Ordina in modo crescente la partizione ed infine controlla se questa è uniforme. Il metodo prosegue semplicemente contando quanti elementi del linspace vi sono tra un elemento della partizione ed il suo successivo, e li valuta usando l'appropriato polinomio, riciclando il vettore del linspace che viene restituito in output.

[Codice MATLAB 4.20Script](#) (script d'innescio)

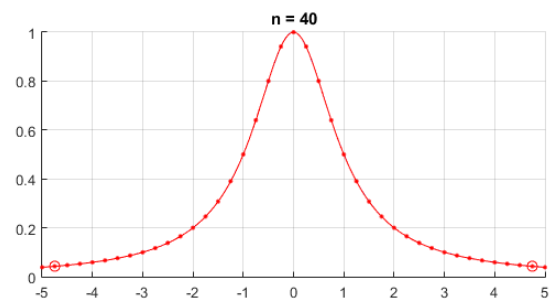
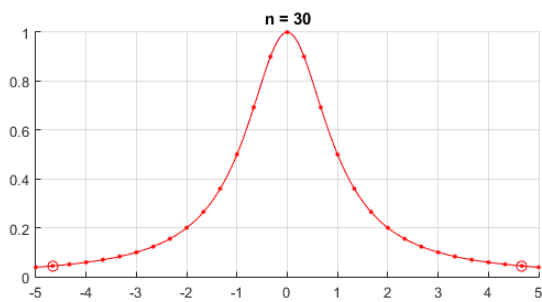
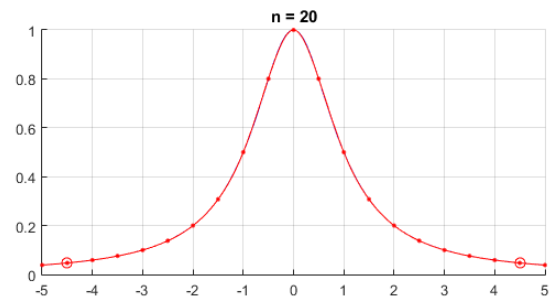
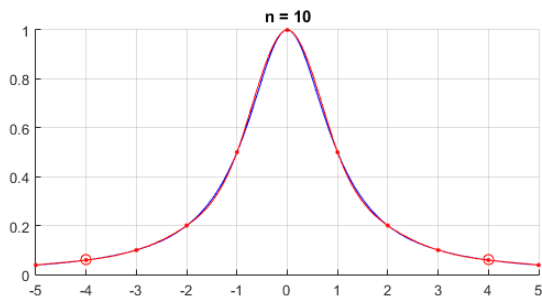
[Codice MATLAB 4.20Grafica](#) (procedura per grafica)

[Codice MATLAB 4.20ErrTab](#) (procedura per tabella)

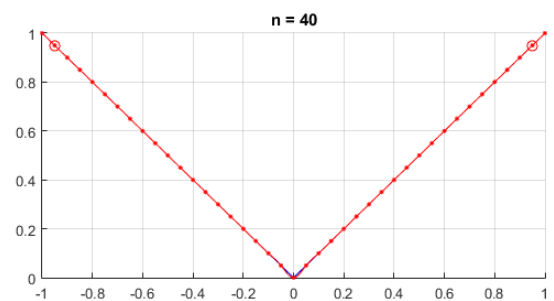
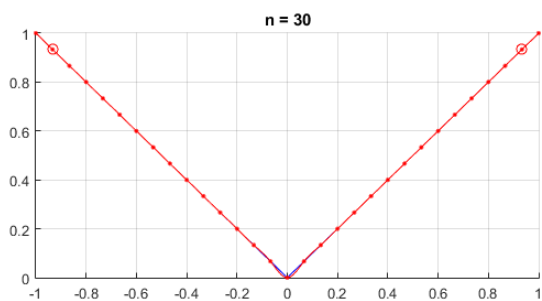
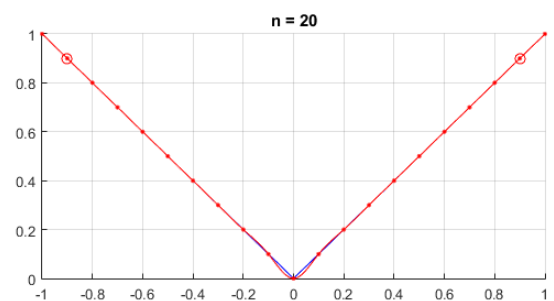
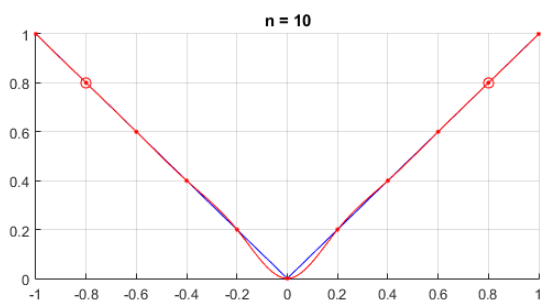
[Codice MATLAB 4.20](#) (funzione per valutazione spline)

Sebbene l'esecuzione dello script d'innescio crei i grafici di tutte le funzioni per tutte le n richieste, presentiamo solo i grafici riguardanti le spline *not a knot* poiché non vi sono sostanziali differenze con quelli per le spline *naturali*. I punti definiti *not a knot* sono evidenziati da un cerchietto.

Esempio di Runge



Esempio di Bernstein



Esempio del pdf di complemento

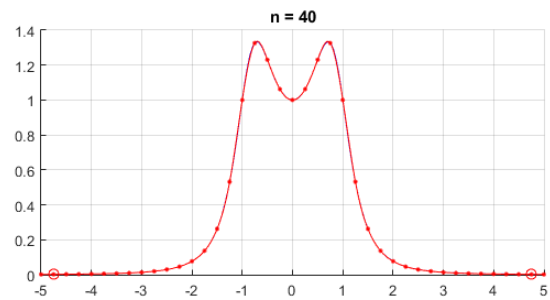
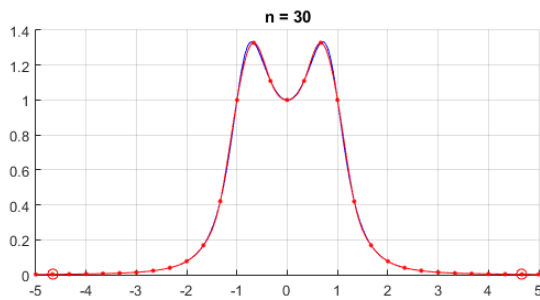
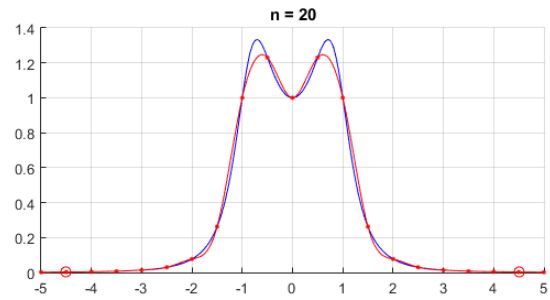
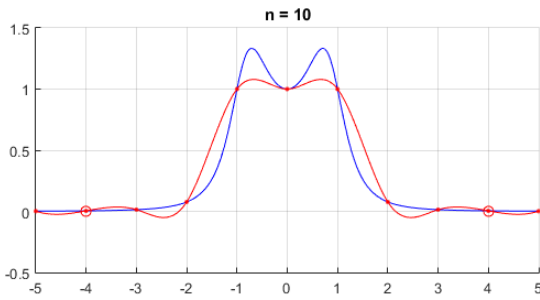


Tabella MATLAB degli errori

	Runge	Bernstein	Pdf
n = 10	$\ e\ = 3.331e-16$, 0.33 s	$\ e\ = 1.665e-16$, 0.36 s	$\ e\ = 6.661e-16$, 0.34 s
n = 20	$\ e\ = 3.331e-16$, 0.43 s	$\ e\ = 2.220e-16$, 0.43 s	$\ e\ = 7.772e-16$, 0.43 s
n = 30	$\ e\ = 5.551e-16$, 0.55 s	$\ e\ = 2.220e-16$, 0.57 s	$\ e\ = 8.882e-16$, 0.56 s
n = 40	$\ e\ = 3.331e-16$, 0.68 s	$\ e\ = 2.220e-16$, 0.69 s	$\ e\ = 8.882e-16$, 0.68 s

Esercizio 4.21

Interpretare la retta $r(x)$ dell'Esercizio 3.32 come retta di approssimazione ai minimi quadrati dei dati.

Svolgimento

La retta è $r(x) = a_1x + a_2$.

I dati sono:

- $x = \text{linspace}(0,10,101)'$
- $\text{gamma} = 0.1$
- $y = 10 * x + 5 + (\sin(x * \text{pi})) * \text{gamma}$

La prima istruzione crea 101 ascisse equispaziate, l'ultima istruzione ci dà le relative ordinate. Quindi abbiamo $n = 101$ coppie di punti del tipo (x_i, y_i) , che costituiscono le nostre misure

sperimentali. Sappiamo che la legge polinomiale che descrive il fenomeno, la nostra $r(x)$, ha grado $m = 1$, quindi è nella forma:

$$r(x) = \sum_{k=0}^m a_k x^k = a_0 + a_1 x \equiv a_1 x + a_2$$

Con riferimento alla (4.63) di pag. 104 del libro di testo, definiamo il vettore z :

$$z = \begin{pmatrix} r(x_1) \\ \vdots \\ r(x_{101}) \end{pmatrix}, \quad z = V \underline{a}$$

Difatti è possibile vedere tale vettore anche come la matrice delle incognite V di Vandermonde (quindi di rango massimo) che moltiplica il vettore \underline{a} dei coefficienti.

Definiamo quindi il vettore residuo come $\underline{y} - \underline{z}$. Quello che vogliamo fare è minimizzarlo, ovvero determinare il vettore dei termini noti \underline{a} che minimizzi la quantità:

$$\|\underline{y} - \underline{z}\|_2^2 = \sum_{i=0}^n |y_i - z_i|^2$$

Si tratta quindi di risolvere il sistema sovradeterminato $V \underline{a} = \underline{y}$:

$$\begin{pmatrix} 1 & x_0 \\ \vdots & \vdots \\ 1 & x_{101} \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \end{pmatrix} = \begin{pmatrix} y_1 \\ \vdots \\ y_{101} \end{pmatrix}$$

Tale sistema ammette soluzione se e solo se la matrice Vandermonde dei coefficienti ha rango massimo, che nel nostro caso equivale ad avere almeno due vettori riga linearmente indipendenti, ovvero basta avere almeno 2 ascisse distinte, cosa indubbiamente vera data l'istruzione `linspace(0,1,101)`. Il sistema potrà quindi essere risolto tramite la fattorizzazione QR della matrice.

Esercizio 4.22

È noto che un fenomeno che ha un decadimento esponenziale modellizzato come

$$y = \alpha e^{-\lambda t}$$

in cui α e λ sono parametri positivi e incogniti. Riformulare il problema in modo che il modello sia di tipo polinomiale. Supponendo inoltre di disporre delle seguenti misure,

t_i	0	1	2	3	4	5	6	7	8	9	10
y_i	5.22	4.00	4.28	3.89	3.53	3.12	2.73	2.70	2.20	2.08	1.94

Calcolare la stima ai minimi quadrati dei due parametri incogniti. Valutare il residuo e raffigurare, infine, i risultati ottenuti.

Svolgimento

Manipoliamo y in modo da portare il problema ad un modello polinomiale, passando ai logaritmi:

$$\log(y) = \log(\alpha) - \log(e^{-\lambda})t$$

dove poniamo:

$$x = \log(y)$$

$$z = \log(\alpha)$$

$$d = -\lambda$$

Il nostro problema di partenza, riformulato in forma polinomiale, è dunque diventato:

$$x = z + dt$$

Andiamo a calcolare la stima di z e d , risolvendo il sistema lineare sovradeterminato:

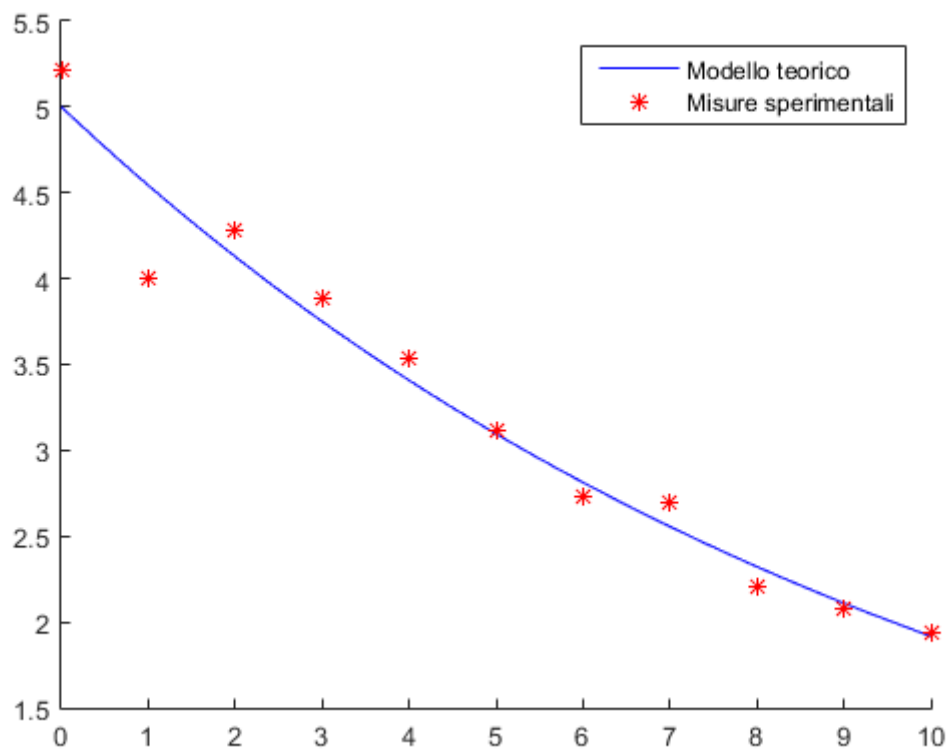
$$\begin{pmatrix} 1 & t_0 \\ 1 & t_1 \\ \vdots & \vdots \\ 1 & t_{10} \end{pmatrix} \begin{pmatrix} z \\ d \end{pmatrix} = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{10} \end{pmatrix}$$

Tale sistema può essere risolto mediante fattorizzazione QR dato che il *rank* della matrice è massimo e dato che abbiamo almeno due ascisse distinte. La matrice dei coefficienti è una matrice del tipo Vandermonde. Presentiamo a seguire il codice per svolgere l'esercizio ed i risultati ottenuti.

Codice MATLAB 4.22

```
Command Window
New to MATLAB? See resources for Getting Started.

Residuo: 0.17084
Alpha: 5.00082
Lambda: 0.09590
fx >> |
```



Esercizio complementare 1

Un corpo si muove di moto uniformemente accelerato. Se ne misura la posizione s a determinati istanti, ottenendo i dati della seguente tabella:

t	$s(t)$
1	6.571
2	13.283
3	23.137
4	36.134
5	52.269
6	71.551
7	93.971
8	119.530
9	148.236
10	180.072

Stimare l'accelerazione e la velocità e posizioni iniziali del corpo.

Svolgimento

Sono note le coppie del tipo (t_i, s_i) ed il modello polinomiale dato dalla legge oraria del moto uniformemente accelerato:

$$x(t) = x_0 + v_0 t + \frac{1}{2} a t^2$$

Possiamo andare a stimare posizione iniziale (data da x_0), la velocità iniziale (data da v_0) e l'accelerazione iniziale (data da a , che sappiamo essere costante per tutto il moto). Risolviamo quindi il problema nel senso dei minimi quadrati:

$$\begin{pmatrix} t_1^0 & t_1 & t_1^2 \\ t_2^0 & t_2 & t_2^2 \\ \vdots & \vdots & \vdots \\ t_{10}^0 & t_{10} & t_{10}^2 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 4 \\ \vdots & \vdots & \vdots \\ 1 & 10 & 100 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \end{pmatrix} = \begin{pmatrix} s_1 \\ s_2 \\ \vdots \\ s_{10} \end{pmatrix}$$

Tale sistema è risolvibile mediante fattorizzazione QR, le soluzioni significano rispettivamente:

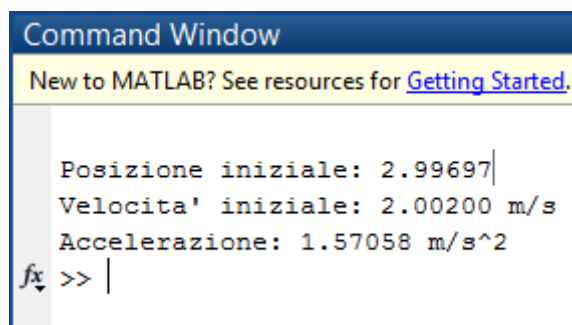
a_0 : posizione iniziale del corpo

a_1 : velocità iniziale del corpo

a_2 : accelerazione iniziale del corpo

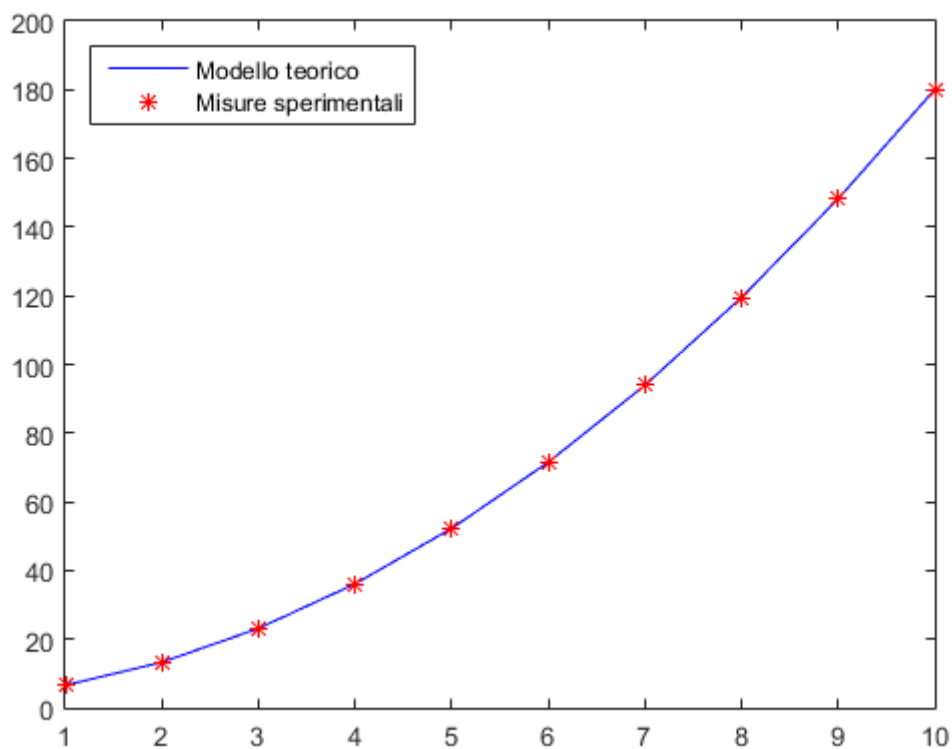
Presentiamo a seguire il codice scritto per svolgere l'esercizio ed i risultati ottenuti.

[Codice MATLAB 4.1 complementare](#)



```
Command Window
New to MATLAB? See resources for Getting Started.

Posizione iniziale: 2.99697|
Velocita' iniziale: 2.00200 m/s
Accelerazione: 1.57058 m/s^2
fx >> |
```



Esercizio complementare 2

Una determinata quantità di materiale decade radioattivamente. La sua attività radioattiva, $r(t)$, è misurata come segue:

t	$r(t)$
1	0.905
2	0.819
3	0.741
4	0.670
5	0.607
6	0.549
7	0.497
8	0.449
9	0.407
10	0.368

Calcolare la costante di decadimento del materiale.

Svolgimento

Lo svolgimento di questo esercizio è analogo a quello visto per l'esercizio 4.22, partendo dal modello:

$$y = \alpha e^{-\lambda t}$$

in cui λ e α sono parametri positivi incogniti, si manipola y in modo da portare il problema ad un modello polinomiale col passaggio ai logaritmi:

$$\log(y) = \log(\alpha) - \log(e^{-\lambda})t$$

dove poniamo:

$$x = \log(y)$$

$$z = \log(\alpha)$$

$$d = -\lambda$$

Il nostro problema di partenza, riformulato in forma polinomiale, è dunque diventato:

$$x = z + dt$$

Andiamo a calcolare la stima di z e d , risolvendo il sistema lineare sovra determinato:

$$\begin{pmatrix} 1 & t_0 \\ 1 & t_1 \\ \vdots & \vdots \\ 1 & t_{10} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \end{pmatrix} = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{10} \end{pmatrix}$$

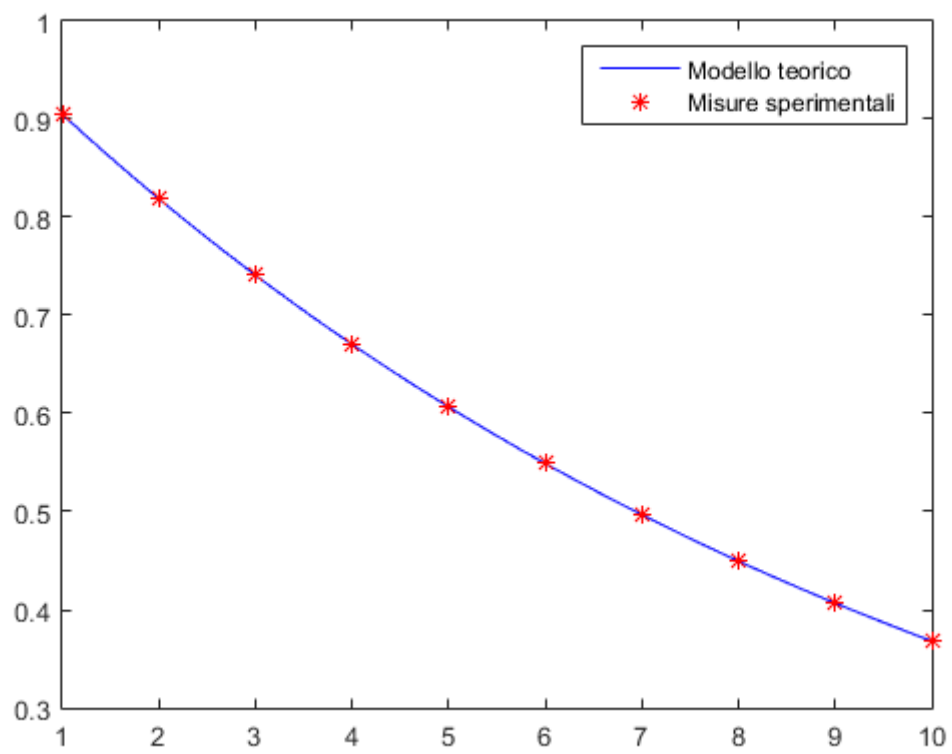
Con $a_0 = z$ e $a_1 = d$. Tale sistema può essere risolto mediante fattorizzazione QR dato che il *rank* della matrice è massimo poiché abbiamo almeno due ascisse distinte. La matrice dei coefficienti è una matrice di tipo Vandermonde. Osserviamo che la costante di decadimento del materiale corrisponde al parametro $-d$ del modello polinomiale.

Presentiamo a seguire il codice scritto per svolgere l'esercizio ed i risultati ottenuti.

[Codice MATLAB 4.2 complementare](#)

```
Command Window
New to MATLAB? See resources for Getting Started.

Alpha: 1.00012
Lambda: 0.09997
fx >> |
```



ESERCIZI CAPITOLO 5

FORMULE DI QUADRATURA

Esercizio 5.1

Calcolare il numero di condizionamento dell'integrale $\int_0^{e^{21}} \sin\sqrt{x} dx$. Questo problema è ben condizionato o è mal condizionato?

Svolgimento

Studiando il condizionamento del problema abbiamo che:

$$\begin{aligned} |I(f) - I(\tilde{f})| &= \left| \int_a^b (f(x) - \tilde{f}(x)) dx \right| \leq \int_a^b |f(x) - \tilde{f}(x)| dx \\ &\leq \|f - \tilde{f}\| \int_a^b dx = (b - a) \|f - \tilde{f}\| \end{aligned}$$

dove $\|f - \tilde{f}\|$ rappresenta i dati perturbati in ingresso e $b - a$ l'ampiezza dell'intervallo, che rappresenta il fattore di condizionamento. Quindi $k = b - a$. Applicando quanto appreso dalla teoria possiamo affermare che $k = e^{21} - 0$, ovvero un ordine di grandezza pari a 10^9 , quindi il problema è molto mal condizionato.

Esercizio 5.2

Derivare, dalla (5.5), i coefficienti della formula dei trapezi (5.6) e della formula di Simpson (5.7).

Svolgimento

L'espressione dei coefficienti c_{kn} (5.5) della formula di quadratura di Newton-Cotes, della quale la formula dei trapezi e di Simpson sono espressioni particolari rispettivamente con $n = 1$ ed $n = 2$, è:

$$c_{kn} = \int_0^n \prod_{\substack{j=0 \\ j \neq k}}^n \frac{t-j}{k-j} dt \quad \text{con } k = 0, \dots, n$$

Con $n = 1$ (formula dei trapezi) abbiamo:

$$c_{0,1} = \int_0^1 \prod_{\substack{j=0 \\ j \neq k}}^1 \frac{t-j}{k-j} dt = \int_0^1 \frac{t-1}{-1} dt = - \int_0^1 (t-1) dt = - \left(\int_0^1 t dt - \int_0^1 dt \right) = - \left(\frac{1}{2} - 1 \right) = \frac{1}{2}$$

$$c_{1,1} = \int_0^1 \prod_{\substack{j=0 \\ j \neq k}}^1 \frac{t-j}{k-j} dt = \int_0^1 \frac{t-0}{1-0} dt = \int_0^1 t dt = \frac{1}{2} - 0 = \frac{1}{2}$$

Per trovare il secondo coefficiente, avremmo potuto sfruttare anche l'enunciato del teorema 5.1:

$$\frac{1}{n} \sum_{k=0}^n c_{kn} = 1$$

Con $n = 2$ (formula di Simpson) abbiamo:

$$\begin{aligned} c_{02} &= \int_0^2 \prod_{\substack{j=0 \\ j \neq k}}^2 \frac{t-j}{k-j} dt = \int_0^2 \frac{t-1}{-1} \cdot \frac{t-2}{-2} dt = \\ &= \frac{1}{2} \int_0^2 (t^2 - 3t + 2) dt = \frac{1}{2} \left(\int_0^2 t^2 dt - 3 \int_0^2 t dt + 2 \int_0^2 dt \right) = \frac{1}{2} \left(\frac{8}{3} - 6 + 4 \right) = \frac{1}{3} \end{aligned}$$

$$c_{12} = \int_0^2 \prod_{\substack{j=0 \\ j \neq k}}^2 \frac{t-j}{k-j} dt = \int_0^2 -t(t-2) dt = - \int_0^2 t^2 dt + 2 \int_0^2 t dt = -\frac{8}{3} + 4 = \frac{4}{3}$$

Sfruttando il teorema 5.1 troviamo c_{22} :

$$\frac{1}{2} \sum_{k=0}^2 c_{k2} = c_{02} + c_{12} + c_{22} = 2$$

$$\begin{aligned} \frac{1}{3} + \frac{4}{3} + c_{22} &= 2 \\ c_{22} &= \frac{1}{3} \end{aligned}$$

Esercizio 5.3

Verificare, utilizzando il risultato del Teorema 5.2, le (5.10) e (5.11).

Svolgimento

Il teorema 5.2 ci dice che se $f(x) \in C^{(n+k)}$, con

$$k = \begin{cases} 1, & \text{se } n \text{ dispari} \\ 2, & \text{se } n \text{ è pari} \end{cases}$$

allora l'errore di quadratura $E_n(f) = I(f) - I_n(f)$ è dato da:

$$E_n(f) = v_n \frac{f^{(n+k)}(\xi)}{(n+k)!} \left(\frac{b-a}{n}\right)^{n+k+1}$$

per un opportuno $\xi \in [a, b]$, dove:

$$v_n = \begin{cases} \int_0^n \prod_{j=0}^n (t-j) dt & \text{se } n \text{ è dispari} \\ \int_0^n t \prod_{j=0}^n (t-j) dt & \text{se } n \text{ è pari} \end{cases}$$

Dalla formula di Newton-Cotes, per $n = 1$ abbiamo la formula dei trapezi. Per il teorema 5.2, $k = 1$. Calcoliamo v_1 :

$$v_1 = \int_0^1 \prod_{j=0}^1 (t-j) dt = \int_0^1 t(t-1) dt = \int_0^1 t^2 dt - \int_0^1 t dt = -\frac{1}{3} - \frac{1}{2} = -\frac{1}{6}$$

Quindi la (5.10) diventa:

$$E_1(f) = -\frac{1}{12} f^{(2)}(\xi)(b-a)^3, \quad \text{con } \xi \in [a, b]$$

Dalla formula di Newton-Cotes, per $n = 2$ abbiamo la formula di Simpson. Per il teorema 5.2, $k = 2$. Calcoliamo v_2 :

$$v_2 = \int_0^2 t \prod_{j=0}^2 (t-j) dt = \int_0^2 t^2(t-1)(t-2) dt = \int_0^2 t^4 dt - 3 \int_0^2 t^3 dt + 2 \int_0^2 t^2 dt = -\frac{4}{15}$$

Quindi la (5.11) diventa:

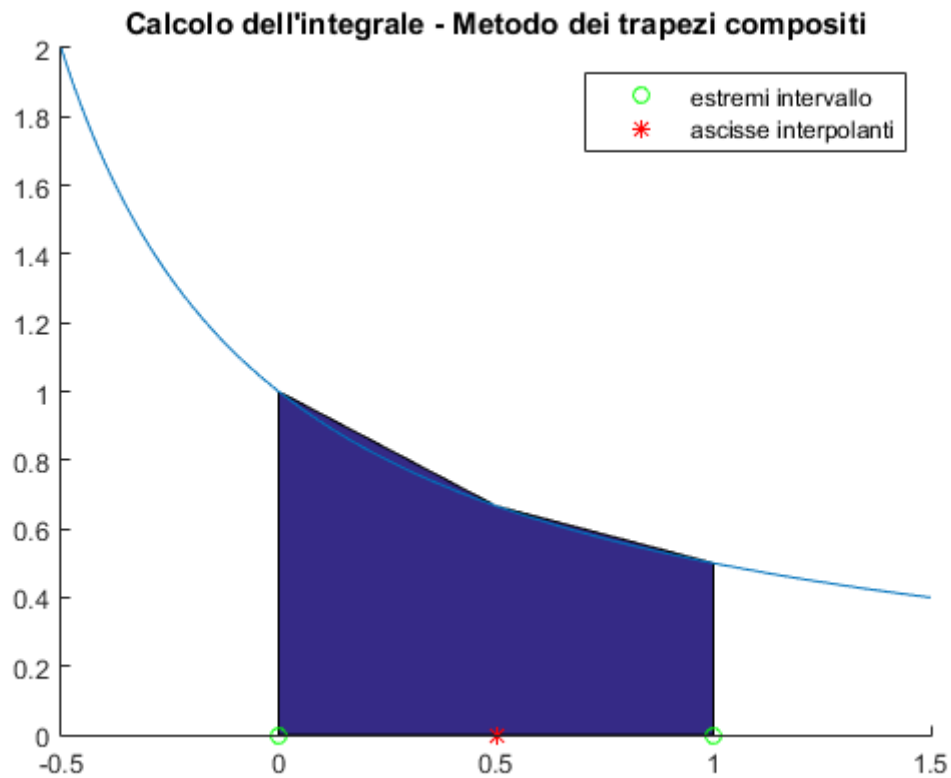
$$E_2(f) = -\frac{1}{90} f^{(4)}(\xi) \left(\frac{b-a}{2}\right)^5, \quad \text{con } \xi \in [a, b]$$

Esercizio 5.4

Scrivere una *function* MATLAB che implementi efficientemente la formula dei trapezi composta (5.12).

Svolgimento

[Codice MATLAB 5.4](#)

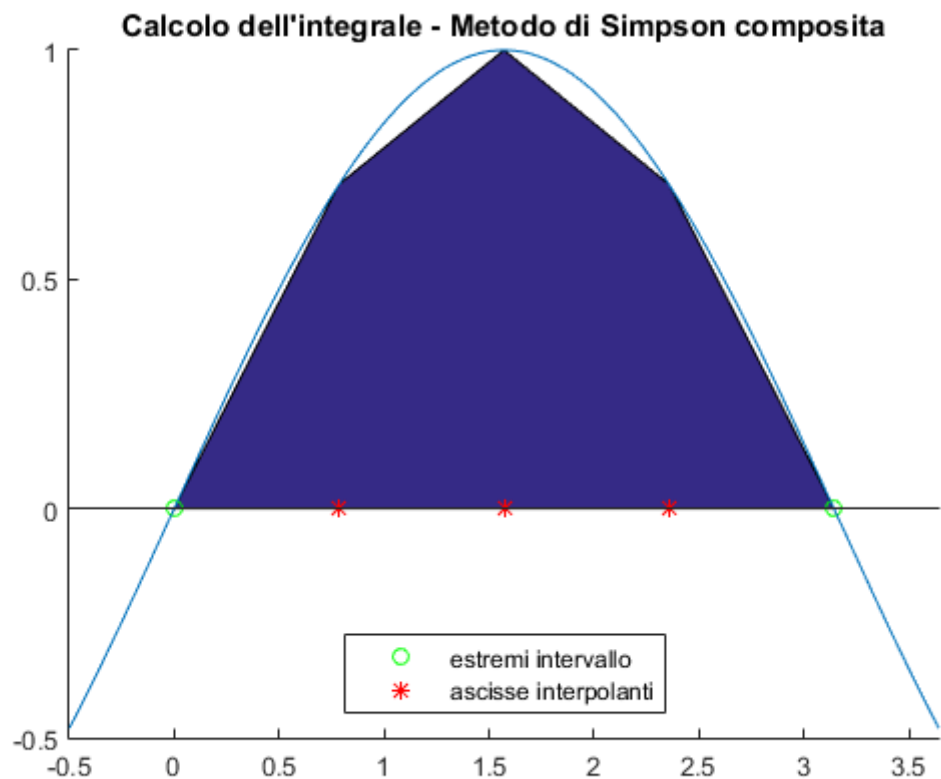


Esercizio 5.5

Scrivere una *function* MATLAB che implementi efficientemente la formula di Simpson composta (5.14).

Svolgimento

Codice MATLAB 5.5



Esercizio 5.6

Implementare efficientemente in MATLAB la formula adattativa dei trapezi.

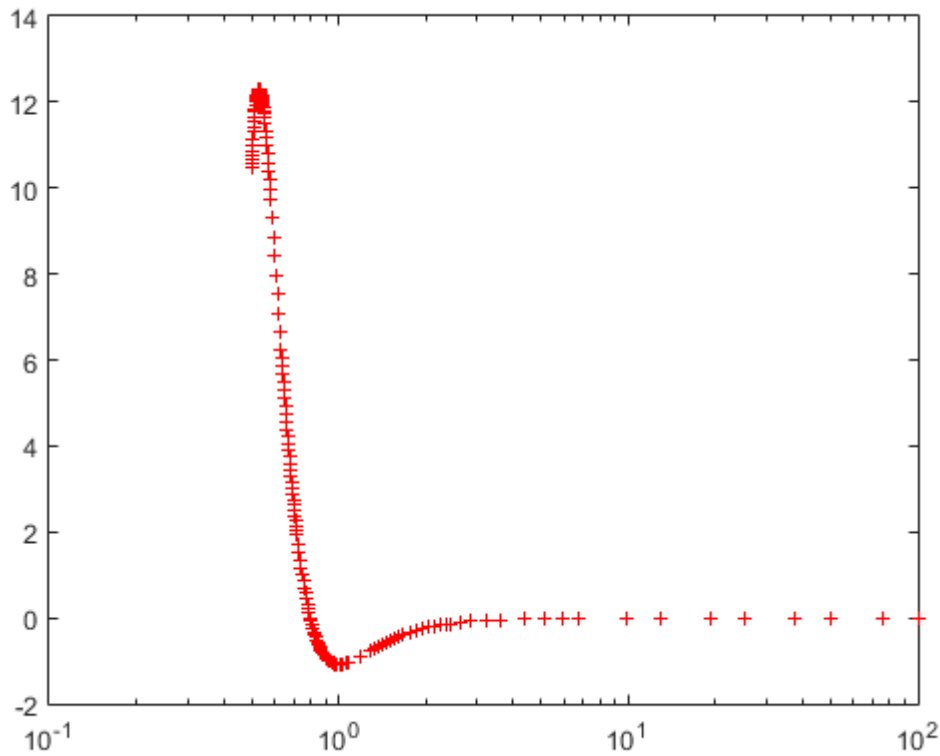
Svolgimento

Prima di presentare il codice mostriamo come si ottiene la variabile I_2 che implementa la formula composta dei trapezi a passo costante, usata nell'algoritmo adattativo. Chiamiamo $h = \frac{b-a}{2}$ ed $m = \frac{a+b}{2}$.

$$I_1 = h(f(a) + f(b)) = hf(a) + hf(b)$$

$$I_1^2 = \frac{1}{2}h(f(a) + f(b) + 2f(m)) =$$

$$\frac{1}{2} \left(\overbrace{hf(a) + hf(b)}^{I_1} + 2hf(m) \right) =$$
$$\frac{1}{2}(I_1 + 2hf(m)) = \frac{1}{2}I_1 + hf(m)$$



Esercizio 5.7

Implementare efficientemente in MATLAB la formula adattativa di Simpson.

Svolgimento

Prima di presentare il codice mostriamo come si ottiene la variabile I_4 che implementa la formula composta di Simpson a passo costante, usata nell'algoritmo adattativo. Chiamiamo $h = \frac{b-a}{6}$ ed $m = \frac{a+b}{2}$.

$$I_2 = h(f(a) + f(b) + 4f(m)) = hf(a) + hf(b) + 4hf(m)$$

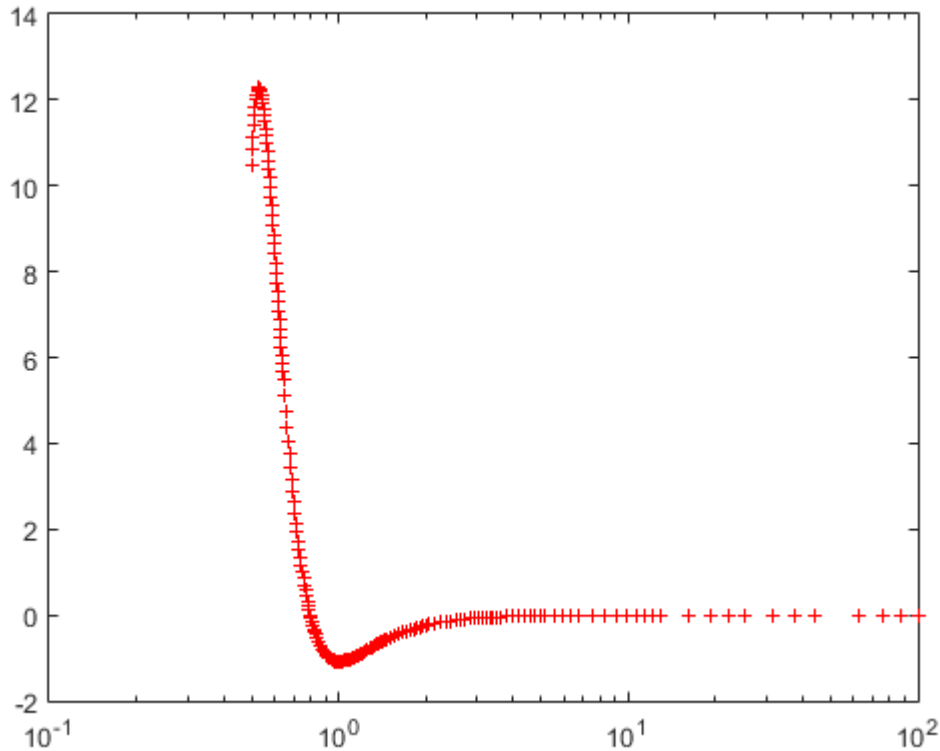
$$I_2^4 = \frac{1}{2}h(f(a) + f(b) + 2f(m) + 4f(x_1) + 4f(x_3)) =$$

$$= \frac{1}{2} \left(\overbrace{hf(a) + hf(b) + 2hf(m)}^{I_2 - 2hf(m)} + 4hf(x_1) + 4hf(x_3) \right) =$$

$$= \frac{1}{2} (I_2 - 2hf(m) + 4hf(x_1) + 4hf(x_3)) =$$

$$= \frac{1}{2} \left(I_2 - 2h(2f(x_1) + 2f(x_3) - f(m)) \right)$$

Codice MATLAB 5.7



Esercizio 5.8

Come è classificabile, dal punto di vista del condizionamento, il problema (5.17)?

Svolgimento

Il problema in esame è:

$$\int_{\frac{1}{2}}^{100} -2x^{-3} \cos(x^{-2}) dx$$

Per classificarlo dal punto di vista del condizionamento calcoliamone il fattore $k = b - a$. Visti i dati del problema, abbiamo che $b = 100$ ed $a = \frac{1}{2}$, quindi $k = 100 - \frac{1}{2} = 99.5$. Tale valore non è sufficientemente alto per poter dire che il problema è mal condizionato.

Esercizio 5.9

Utilizzare le *function* relative agli Esercizi 5.4 e 5.6 per riempire i campi della seguente tabella, riferita all'approssimazione di (5.17).

Formula composta dei trapezi		Formula dei trapezi adattativa		
n	$E_1^{(n)}$	tol	$errore$	$numero\ punti$
1000		10^{-1}		
2000		10^{-2}		
\vdots		10^{-3}		
10000		10^{-4}		

Svolgimento

Formula composta dei trapezi		Formula dei trapezi adattativa		
n	$E_1^{(n)}$	tol	$errore$	$numero\ punti$
1000	$9.28965 * 10^{-2}$	10^{-1}	$5.46956 * 10^{-3}$	159
2000	$2.61305 * 10^{-2}$	10^{-2}	$1.26756 * 10^{-3}$	417
3000	$1.18364 * 10^{-2}$	10^{-3}	$3.30045 * 10^{-4}$	1567
4000	$6.70067 * 10^{-3}$	10^{-4}	$6.59363 * 10^{-5}$	4851
5000	$4.30090 * 10^{-3}$			
6000	$2.99141 * 10^{-3}$			
7000	$2.19984 * 10^{-3}$			
8000	$1.68527 * 10^{-3}$			
9000	$1.33213 * 10^{-3}$			
10000	$1.07934 * 10^{-3}$			

Lo script che presentiamo stampa sulla finestra di comando i valori contenuti nella tabella su riportata:

[Codice MATLAB 5.9](#)

Esercizio 5.10

Utilizzare le *function* relative agli Esercizi 5.5 e 5.7 per riempire i campi della seguente tabella, riferita all'approssimazione di (5.17).

Formula composta di Simpson		Formula di Simpson adattativa		
n	$E_2^{(n)}$	tol	$errore$	$numero\ punti$
1000		10^{-1}		
2000		10^{-2}		
\vdots		10^{-3}		
9000		10^{-4}		
10000		10^{-5}		

Svolgimento

Formula composta di Simpson		Formula di Simpson adattativa		
n	$E_2^{(n)}$	tol	$errore$	$numero\ punti$
1000	$5.55796 * 10^{-2}$	10^{-1}	$1.11642 * 10^{-4}$	49
2000	$3.87525 * 10^{-3}$	10^{-2}	$1.93841 * 10^{-4}$	65
3000	$7.29772 * 10^{-4}$	10^{-3}	$4.80678 * 10^{-6}$	93
4000	$2.24032 * 10^{-4}$	10^{-4}	$1.78079 * 10^{-5}$	181
5000	$9.02086 * 10^{-5}$	10^{-5}	$4.83366 * 10^{-6}$	309
6000	$4.30622 * 10^{-5}$			
7000	$2.30940 * 10^{-5}$			
8000	$1.34787 * 10^{-5}$			
9000	$8.38923 * 10^{-6}$			
10000	$5.49207 * 10^{-6}$			

Lo script che presentiamo stampa sulla finestra di comando i valori contenuti nella tabella su riportata:

[Codice MATLAB 5.10](#)

Esercizio complementare

Si calcoli l'integrale definito tra -1 e 1 della funzione:

$$f(x) = \exp(-(20x)^2)$$

con una tolleranza $tol = 10^{-6}$, utilizzando:

1. La formula composta dei trapezi
2. La formula composta di Simpson
3. La formula adattativa dei trapezi
4. La formula adattativa di Simpson

implementando una stima dell'errore. Calcolare il numero di valutazioni richieste in ciascun caso.

Svolgimento

Abbiamo scritto uno script che lancia le *function* interessate e stampa i risultati. Abbiamo quindi implementato le formule composte a passo costante. Presentiamo codici e risultati:

[Codice MATLAB 5 complementare script](#)

[Codice MATLAB 5 complementare Trapezi](#)

[Codice MATLAB 5 complementare Simpson](#)

Command Window

```
Area trapezi adattativa = 0.0886227772997696  
Valutazioni di funzioni = 2233  
Stima dell'errore = 8.48e-08
```

```
Area Simpson adattativa = 0.0886227279936365  
Valutazioni di funzioni = 145  
Stima dell'errore = 3.54e-08
```

```
Area trapezi composta = 0.0886226925452758  
Valutazioni di funzioni = 129  
Stima dell'errore = 2.97e-15
```

```
Area Simpson composta = 0.0886328650713169  
Valutazioni di funzioni = 131075  
Stima dell'errore = 1.02e-05
```

```
 $f_x$  >> |
```

ESERCIZI CAPITOLO 6

UNA APPLICAZIONE: IL CALCOLO DEL GOOGLE PAGERANK

Esercizio 6.1

(Teorema di Gershgorin) Dimostrare che gli autovalori di una matrice $A = (a_{ij}) \in \mathbb{C}^{n \times n}$ sono contenuti nell'insieme:

$$\mathcal{D} = \bigcup_{i=1}^n \mathcal{D}_i, \quad \mathcal{D}_i = \left\{ \lambda \in \mathbb{C} : |\lambda - a_{ii}| \leq \sum_{\substack{j=1 \\ j \neq i}}^n |a_{ij}| \right\}, \quad i = 1, \dots, n$$

Svolgimento

Prendiamo un λ autovalore di A ed il suo corrispondente autovettore \underline{x} . Vale quindi:

$$A\underline{x} = \lambda\underline{x}$$

Esprimendo il sistema per colonne abbiamo:

$$\sum_{j=1}^n a_{ij}x_j = \lambda x_i, \quad i = 1, \dots, n$$

Ora scegliamo una i t. c. $|x_i| = \max_j |x_j|$, ovvero scegliamo la i in modo tale che x_i sia la più grande componente in valore assoluto del vettore \underline{x} , dove ovviamente $|x_i| > 0$.

Scomponiamo la sommatoria tirando fuori tale x_i scelto:

$$\sum_{j \neq i}^n a_{ij}x_j = \lambda x_i - a_{ii}x_i$$

Passando ai moduli, e dividendo entrambi i membri per $x_i > 0$ scelto, otteniamo:

$$|\lambda - a_{ii}| = \left| \frac{\sum_{j \neq i}^n a_{ij}x_j}{x_i} \right| \leq \sum_{j \neq i}^n |a_{ij}| \left| \frac{x_j}{x_i} \right| \leq \sum_{j \neq i}^n |a_{ij}|$$

dove l'ultima disuguaglianza è vera in quanto abbiamo scelto i in modo che:

$$\left| \frac{x_j}{x_i} \right| \leq 1, \quad i \neq j$$

Esercizio 6.2

Utilizzare il metodo delle potenze per approssimare l'autovalore dominante della matrice

$$A_n = \begin{pmatrix} 2 & -1 & & \\ -1 & 2 & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 2 \end{pmatrix} \in \mathbb{R}^{n \times n}$$

per valori crescenti di n . Verificare numericamente che questo è dato da $2 \left(1 + \cos \frac{\pi}{n+1}\right)$

Svolgimento

Osserviamo preliminarmente che la matrice A_n , all'aumentare di n , diventa sempre più sparsa. Pertanto abbiamo pensato di implementare algoritmi che usano la compressione (per righe in questo caso) della matrice. Per svolgere l'esercizio utilizziamo 4 *function* ed uno script. La *function* 6.2.3 crea la matrice $A_n \in \mathbb{R}^{n \times n}$ come specificato dalla traccia, ma comprimendola, richiamando la *function* 6.2.4. Un accorgimento usato nella 6.2.4 è stato lo stabilire a priori la dimensione della matrice compressa (e quindi anche della relativa matrice COL) in modo da non avere una crescita dinamica della dimensione, azzerando quindi l'overhead che altrimenti si avrebbe per via della riallocazione dinamica dei nuovi blocchi. La *function* 6.2.2 calcola l'autovalore dominante col metodo delle potenze appositamente modificato, richiamando la 6.2.5 facendo il MATVEC per colonne. Lo script itera il procedimento partendo da $n = 3$ aumentandolo di 3 ad ogni iterazione. Mostriamo codici e tabella dei risultati con tolleranze di 10^{-5} e 10^{-10} .

[Codice MATLAB 6.2.1](#) (script)

[Codice MATLAB 6.2.2](#) (metodo delle potenze modificato)

[Codice MATLAB 6.2.3](#) (funzione che crea la matrice secondo la traccia, ma compressa)

[Codice MATLAB 6.2.4](#) (compressione per righe della matrice)

[Codice MATLAB 6.2.5](#) (MATVEC per colonne)

Tolleranza di 10^{-5}

n	Metodo Delle Potenze λ	$2 \left(1 + \cos \left(\frac{\pi}{n+1}\right)\right)$	Errore
3	3.414209	3.414214	$\approx 10^{-6}$
6	3.801914	3.801938	$\approx 10^{-5}$
9	3.902085	3.902113	$\approx 10^{-5}$
12	3.941776	3.941884	$\approx 10^{-4}$
15	3.961410	3.961571	$\approx 10^{-4}$
18	3.972489	3.972723	$\approx 10^{-4}$
21	3.979605	3.979643	$\approx 10^{-5}$
24	3.983816	3.984229	$\approx 10^{-4}$
27	3.986895	3.987424	$\approx 10^{-4}$
30	3.989086	3.989739	$\approx 10^{-4}$

Tolleranza di 10^{-10}

n	Metodo Delle Potenze λ	$2 \left(1 + \cos \left(\frac{\pi}{n+1} \right) \right)$	Errore
3	3.4142135623	3.4142135624	$\approx 10^{-11}$
6	3.8019377355	3.8019377358	$\approx 10^{-10}$
9	3.9021130320	3.9021130326	$\approx 10^{-10}$
12	3.9418836338	3.9418836349	$\approx 10^{-9}$
15	3.9615705592	3.9615705608	$\approx 10^{-9}$
18	3.9727226045	3.9727226068	$\approx 10^{-9}$
21	3.9796428806	3.9796428838	$\approx 10^{-9}$
24	3.9842293985	3.9842294026	$\approx 10^{-9}$
27	3.9874244146	3.9874244198	$\approx 10^{-9}$
30	3.9897386404	3.9897386468	$\approx 10^{-9}$

Osserviamo che, al crescere della dimensione del problema n , l'errore $\left| \lambda - 2 \left(1 + \cos \left(\frac{\pi}{n+1} \right) \right) \right|$ tende, seppur molto lentamente, ad aumentare.

Esercizio 6.3

Dimostrare i corollari 6.2 e 6.3.

Svolgimento

Il corollario 6.2 afferma che:

Se $A = \alpha(I - B)$ è una M-matrice e $A = M - N$, con $0 \leq N \leq \alpha B$, allora M è nonsingolare e lo splitting è regolare. Pertanto il metodo iterativo $M\underline{x}_k = N\underline{x}_{k-1} + \underline{b}$ con $k \geq 1$ è convergente.

Dimostrazione.

Per dimostrare che il metodo è convergente, lo splitting deve essere regolare, ovvero deve soddisfare le condizioni (6.23) di pag. 127: dato uno splitting $A = M - N$, deve essere $M^{-1} \geq 0$ e $N \geq 0$. Per ipotesi sappiamo che $N \geq 0$, quindi ci rimane da dimostrare che $M^{-1} \geq 0$, ovvero che M sia monotona. Sapendo che le M-matrici sono particolari casi di matrici monotone, andiamo a dimostrare che M è una M-matrice:

$$A = M - N \rightarrow M = A + N = \alpha(I - B) + N = \alpha I - \alpha B + N = \alpha I - \alpha \left(B - \frac{1}{\alpha} N \right)$$

Ponendo ora $C = B - \frac{1}{\alpha} N$ giungiamo infine a:

$$M = \alpha(I - C)$$

Quindi affinché M sia una M-matrice rimanere da verificare che $C \geq 0$ e $\rho(C) < 1$

- Dato che $\alpha B \geq N$, allora $B \geq \frac{1}{\alpha} N$, quindi $C \geq 0$
- Dato che $\alpha > 0$, allora $B > C$, e per il Lemma 6.2 si ha che $\rho(C) < \rho(B) < 1$

Il corollario 6.3 afferma che:

se A è una M-matrice e la matrice M in (6.19) è ottenuta ponendo a 0 elementi extradiagonali di A , allora lo splitting (6.19) è regolare. Pertanto, il metodo iterativo (6.20) è convergente.

Dimostrazione:

sia A una M-matrice con uno splitting $A = M - N$, con M matrice diagonale costituita dagli elementi della diagonale principale di A . Allora $A - M = B$ differisce da A solo per gli elementi sulla diagonale principale, che sono nulli. Poiché $A = \alpha(I - B) = \alpha I - \alpha B = M - N$, risulta $\alpha B = N$. In virtù del precedente corollario, se ne deduce che lo splitting è regolare e quindi, per il teorema 6.7, che il metodo iterativo è convergente.

Esercizio 6.4

Dimostrare il teorema 6.9.

Svolgimento

Il teorema dice che, data una matrice A partizionabile in $A = D - L - U$ dove:

- D è diagonale
- L è strettamente triangolare inferiore
- U è strettamente triangolare superiore

se tale matrice è una M-matrice, allora $D, L, U \geq 0$. In particolare, D ha elementi diagonali positivi.

Dimostrazione:

una delle possibili definizioni di M-matrice è la seguente: sia $A \in \mathbb{R}^{n \times n}$ con componenti $a_{ij} \leq 0 \forall i \neq j$ con $i, j = 1, \dots, n$. Allora A è una M-matrice se può essere espressa nella forma $A = \alpha I - B$ con B a componenti $b_{ij} \geq 0$ con $i, j = 1, \dots, n$, con $\rho(B) < \alpha$ ed I la matrice identità. La proprietà che ci interessa ora delle M-matrici è la seguente:

- $a_{ij} \leq 0 \forall i \neq j$
- $a_{ii} > 0$ per $i = 1, \dots, n$

Dimostriamo rapidamente questa proprietà. Se fosse:

$$a_{ii} \leq 0 \rightarrow \begin{pmatrix} a_{1i} \\ a_{2i} \\ \vdots \\ a_{ni} \end{pmatrix} \leq 0$$

ma l' i -esima colonna di A la possiamo scrivere anche come $A\bar{e}_i$, ciò significherebbe che $A\bar{e}_i \leq 0 \rightarrow \bar{e}_i \leq A^{-1}0 = 0$ che è assurdo.

Dimostrata tale struttura, la tesi ne è una diretta conseguenza. Difatti, essendo $D = \text{diag}(A)$ essa ha componenti positive. L contiene la parte strettamente triangolare inferiore di A , ma con i segni cambiati. Discorso analogo vale per U .

Esercizio 6.5

Tenendo conto della (6.15), riformulare il metodo delle potenze (6.16) per il calcolo del Google pagerank come metodo iterativo definito da uno splitting regolare.

Svolgimento

Dato il problema $A\bar{x} = \bar{b}$ con A matrice sparsa $\in \mathbb{R}^{n \times n}$, per il pagerank di Google abbiamo:

$$A = I - pS, \quad S \geq 0, \quad \rho(S) = 1, \quad 0 < p < 1$$

Cerchiamo una scomposizione di A di questo tipo:

$$A = M - N$$

dove A ed M sono nonsingolari. $M - N$ definisce uno splitting della matrice A . Nel sistema $A = I - pS$ associamo $M = I$, $N = pS$, $\bar{b} = \frac{1-p}{n}\bar{e}$. Avremo quindi:

$$(M - N)\bar{x} = \bar{b}, \quad M\bar{x} = N\bar{x} + \bar{b}$$

Giustificiamo ora tali associazioni. Partiamo da:

$$\bar{x} = S(p)\bar{x} = (pS + (1-p)\bar{v}\bar{e}^T)\bar{x}$$

dove $\bar{e} = (1, \dots, 1)$ e $\bar{v} = \frac{1}{n}\bar{e}$. Giungiamo a:

$$\bar{x} = pS\bar{x} + \frac{1-p}{n}\bar{e}\bar{e}^T\bar{x}$$

Poiché $\bar{e}^T\bar{x}_i = \|\bar{x}_i\|_1 = 1$, allora

$$\bar{x} = pS\bar{x} + \frac{1-p}{n}\bar{e}$$

Si giunge infine a:

$$(I - pS)\bar{x} = \frac{1-p}{n}\bar{e} \equiv \bar{b}$$

Possiamo ora definire la procedura iterativa. Assegnato un \bar{x}_0 iniziale, questa sarà:

$$M\bar{x}_{k+1} = N\bar{x}_k + \bar{b}, \quad k = 0, 1, \dots$$

Ovvero:

$$\underline{x}_{k+1} = pS\underline{x}_k + \frac{1-p}{n}\underline{e}$$

La convergenza è assicurata dal fatto che $\rho(I^{-1}N) = \rho(N) < 1$

Esercizio 6.6

A complemento di quanto riportato nell'osservazione 6.4, dimostrare che il metodo di Jacobi converge asintoticamente in un numero minore di iterazioni, rispetto al metodo delle potenze (6.16) per il calcolo del Google pagerank.

Svolgimento

Il metodo iterativo di Jacobi converge asintoticamente più velocemente rispetto al metodo delle potenze per il calcolo del Google pagerank in quanto il raggio spettrale della sua matrice è strettamente minore di 1, mentre quello della matrice del Google pagerank è uguale a 1. Lo si dimostra facendo l'analisi dell'errore. Per il metodo delle potenze abbiamo:

$$e_{k+1} = S(p)e_k \rightarrow \|e_k\| \leq \|S(p)\| \|e_k\| \rightarrow \|e_k\| \leq \|S(p)\|^k \|e_0\|$$

Per il metodo di Jacobi abbiamo invece:

$$e_{k+1} = M^{-1}Ne_k \rightarrow \|e_k\| \leq \|M^{-1}N\| \|e_k\| \rightarrow \|e_k\| \leq \|(M^{-1}N)^k\| \|e_0\|$$

Dal confronto dei raggi spettrali delle due matrici $S(p)$ e $M^{-1}N$ si osserva che:

$$\rho(M^{-1}N) < 1, \quad \rho(S(p)) = 1$$

Pertanto, per $k \rightarrow \infty$, ovvero asintoticamente, il metodo di Jacobi convergerà in meno iterazioni rispetto al metodo delle potenze.

Esercizio 6.7

Dimostrare che, se A è diagonale dominante, per riga o per colonna, il metodo di Jacobi è convergente.

Svolgimento

Sia $A \in \mathbb{C}^{n \times n}$ a componenti a_{ij} . Sappiamo che $A = M - N = D - L - U$. Lo splitting di Jacobi è definito come $A = D - (L + U)$, dove quindi $D = M$ e $N = L + U$. Chiamiamo ora K la matrice di iterazione di Jacobi, pertanto $K = M^{-1}N$. Andiamo a definire K :

$$K = \begin{pmatrix} \frac{1}{a_{11}} & & & \\ & \ddots & & \\ & & \ddots & \\ & & & \frac{1}{a_{nn}} \end{pmatrix} \begin{pmatrix} 0 & a_{12} & \cdots & a_{1n} \\ a_{21} & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & a_{n-1,n} \\ a_{n1} & \cdots & a_{n,n-1} & 0 \end{pmatrix} = \begin{pmatrix} 0 & \frac{a_{12}}{a_{11}} & \cdots & \frac{a_{1n}}{a_{11}} \\ \frac{a_{21}}{a_{22}} & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \frac{a_{n-1,n}}{a_{n-1,n-1}} \\ \frac{a_{n1}}{a_{nn}} & \cdots & \frac{a_{n,n-1}}{a_{nn}} & 0 \end{pmatrix}$$

Sfruttiamo ora le conclusioni del teorema di Gershgorin, che ci dice che ogni autovalore λ di K appartiene all'insieme \mathcal{D} così definito:

$$\mathcal{D} = \bigcup_{i=1}^n \mathcal{D}_i, \quad \mathcal{D}_i = \left\{ \lambda \in \mathbb{C} : |\lambda - k_{ii}| \leq \sum_{\substack{j=1 \\ j \neq i}}^n |k_{ij}| \right\}, \quad i = 1, \dots, n$$

Considerando come è costruita K , risulta quindi che:

$$\mathcal{D}_i = \left\{ \lambda \in \mathbb{C} : |\lambda| \leq \sum_{\substack{j=1 \\ j \neq i}}^n \left| \frac{a_{ij}}{a_{ii}} \right| \right\}, \quad i = 1, \dots, n$$

Ora: se A è a diagonale dominante per righe significa che:

$$|\lambda| \leq \frac{1}{|a_{ii}|} \sum_{\substack{j=1 \\ j \neq i}}^n |a_{ij}| \leq 1$$

e questo è vero per ogni \mathcal{D}_i . Inoltre non importa se A è a diagonale dominante per righe o per colonne, poiché $\lambda(A) = \lambda(A^T)$. Risulta pertanto che $\rho(K) < 1$, ovvero che il metodo è convergente.

Esercizio 6.8

Dimostrare che, se A è diagonale dominante, per riga o per colonna, il metodo di Gauss-Seidel è convergente.

Svolgimento

Il lemma 3.6 di pag. 56 del libro di testo dice che, se una matrice $A \in \mathbb{R}^{n \times n}$ è diagonale dominante per righe (o per colonne), allora è nonsingolare. Teniamo questo risultato a mente ed andiamo ad analizzare lo splitting di Gauss-Seidel: $A = M - N = D - L - U = (D - L) - U$. Sappiamo che, affinché il metodo sia convergente, deve essere $\rho(M^{-1}N) < 1$, ovvero avremo

$\lambda \in \sigma(M^{-1}N)$ tale che $\det(M^{-1}N - \lambda I) = 0$. Manipoliamo quest'ultima espressione sfruttando le numerose proprietà dei determinanti e del prodotto matriciale:

$$\begin{aligned} 0 &= \det(M^{-1}N - \lambda I) = \det(M^{-1}N - \lambda IMM^{-1}) = \det(M^{-1}(N - \lambda M)) = \\ &= \det(M^{-1})\det(N - \lambda M) = \det(M^{-1})\det(\lambda M - N) = 0 \end{aligned}$$

Dal momento che, per definizione di splitting, $\det(M) \neq 0$ (vedi (6.20) pag. 126), allora anche $\det(M^{-1}) \neq 0$. Questo significa che a fare 0 sarà il determinante della matrice $\lambda M - N$, ovvero che tale matrice è singolare. Supponiamo per assurdo allora che $|\lambda| \geq 1$. Per costruzione, tale matrice avrà questa forma:

$$\begin{pmatrix} \lambda a_{11} & a_{12} & \cdots & a_{1,n-1} & a_{1n} \\ \lambda a_{21} & \lambda a_{22} & \ddots & a_{2,n-1} & a_{2n} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \lambda a_{n-1,1} & \lambda a_{n-1,2} & \ddots & \ddots & a_{n-1,n} \\ \lambda a_{n1} & \lambda a_{n2} & \cdots & \lambda a_{n,n-1} & \lambda a_{nn} \end{pmatrix}$$

Sempre per costruzione, così come A è a diagonale dominante, così risulta essere questa matrice. Ma se una matrice è a diagonale dominante, allora è nonsingolare. Siamo quindi giunti all'assurdo. Allora dovrà essere $|\lambda| < 1$, e quindi se A è a diagonale dominante, il metodo di Gauss-Seidel converge.

Esercizio 6.9

Se A è *sdp*, il metodo di Gauss – Seidel risulta essere convergente. Dimostrare questo risultato nel caso (assai più semplice) in cui l'autovalore di massimo modulo della matrice di iterazione sia reale.

(Suggerimento: considerare il sistema lineare equivalente)

$$\left(D^{-\frac{1}{2}}AD^{-\frac{1}{2}}\right)\left(D^{\frac{1}{2}}x\right) = \left(D^{-\frac{1}{2}}b\right), \quad D^{\frac{1}{2}} = \text{diag}(\sqrt{a_{11}}, \dots, \sqrt{a_{nn}}),$$

la cui matrice dei coefficienti è ancora *sdp* ma ha diagonale unitaria, ovvero del tipo $I - L - L^T$. Osservare quindi che, per ogni vettore reale v di norma 1, si ha (vedere la Sezione A.1):

$$v^T L v = v^T L^T v = \frac{1}{2} v^T (L + L^T) v < \frac{1}{2}$$

Svolgimento

Consideriamo il sistema lineare equivalente ad $A\underline{x} = \underline{b}$ del suggerimento, e chiamiamo $C = \left(D^{-\frac{1}{2}}AD^{-\frac{1}{2}}\right)$. Tale matrice è ancora a diagonale unitaria:

$$c_{ii} = \frac{1}{\sqrt{a_{ii}}} a_{ii} \frac{1}{\sqrt{a_{ii}}} = 1$$

Inoltre C è ancora sdp , ovvero del tipo $C = I - L - L^T$. Per definizione di sdp , sappiamo che per ogni $\underline{v} \neq 0$ vale $\underline{v}^T A \underline{v} > 0$. Avremo quindi:

$$\underline{v}^T \underline{v} > \underline{v}^T L \underline{v} + \underline{v}^T L^T \underline{v} \rightarrow \underline{v}^T L \underline{v} = \underline{v}^T L^T \underline{v} < \frac{1}{2}$$

Assumiamo ora $|\lambda| \in \mathbb{R}$ come il raggio spettrale della matrice di iterazione dello splitting di Gauss-Seidel, ovvero *t. c.* $|\lambda| = \rho(M_{GS}^{-1} N_{GS}) = \rho((I - L)^{-1} L^T)$, con \underline{v} relativo autovettore.

Pertanto avremo:

$$(I - L)^{-1} L^T = \lambda \underline{v} \rightarrow \lambda \underline{v} = L^T \underline{v} + \lambda L \underline{v}$$

Quindi:

$$\lambda = \underline{v}^T L \underline{v} + \lambda \underline{v}^T L \underline{v} = ((1 + \lambda) \underline{v}^T L) \underline{v}$$

Da cui ricaviamo:

$$\frac{\lambda}{1 + \lambda} = \underline{v}^T L \underline{v} < \frac{1}{2} \rightarrow -1 < \lambda < 1$$

Pertanto il raggio spettrale della matrice d'iterazione del metodo di Gauss-Seidel è minore stretto di 1.

Esercizio 6.10

Con riferimento ai vettori errore (6.21) e residuo (6.25) dimostrare che, se

$$\|r_k\| \leq \varepsilon \|b\|$$

allora

$$\|e_k\| \leq \varepsilon \kappa(A) \|\hat{x}\|$$

dove $\kappa(A)$ denota, al solito, il numero di condizionamento della matrice A . Concludere che, per sistemi lineari malcondizionati, anche la risoluzione iterativa (al pari di quella diretta) risulta essere più problematica.

Svolgimento

Ricordiamo le definizioni delle (6.21) e (6.25) presenti sul libro di testo:

- $e_k = x_k - \hat{x}$
- $r_k = Ax_k - b$

E ricordando che $\kappa(A) = \|A^{-1}\| \|A\|$, avremo quindi:

$$\underline{r}_k = A \underline{x}_k - \underline{b} = A \underline{x}_k - A \underline{\hat{x}} = A(\underline{x}_k - \underline{\hat{x}}) = A \underline{e}_k$$

Dato che A è nonsingolare, moltiplichiamo entrambi i membri per la sua inversa, ottenendo $\underline{e}_k = A^{-1}\underline{r}_k$. Ricordandoci dell'ipotesi $\|r_k\| \leq \varepsilon\|b\|$, passando alle norme abbiamo:

$$\|e_k\| = \|A^{-1}r_k\| \leq \|A^{-1}\|\|r_k\| \leq \|A^{-1}\|\varepsilon\|b\| = \varepsilon\|A^{-1}\|\|A\|\|\hat{x}\| \leq \varepsilon\|A^{-1}\|\|A\|\|\hat{x}\| = \varepsilon\kappa(A)\|\hat{x}\|$$

che è la tesi, da cui risulta:

$$\frac{\|e_k\|}{\|\hat{x}\|} \leq \varepsilon\kappa(A)$$

dove è evidente come, anche per i metodi iterativi, il sistema è ben condizionato per $\varepsilon\kappa(A) \approx 1$ e malcondizionato quando $\varepsilon\kappa(A) \gg 1$. Notiamo inoltre che all'aumentare del fattore di condizionamento $\kappa(A)$, la nostra tolleranza ε diventa più stretta.

Esercizio 6.11

Calcolare il polinomio caratteristico della matrice:

$$\begin{pmatrix} 0 & \cdots & 0 & \alpha \\ 1 & \ddots & & 0 \\ & \ddots & \ddots & \vdots \\ 0 & & 1 & 0 \end{pmatrix} \in \mathbb{R}^{n \times n}$$

Svolgimento

Determinare il polinomio caratteristico di una matrice data A significa calcolare il $\det(A - \lambda I)$, dove I è la matrice identità. Dovremo quindi calcolare il determinante di una matrice così fatta:

$$\begin{pmatrix} -\lambda & \cdots & 0 & \alpha \\ 1 & \ddots & & 0 \\ & \ddots & \ddots & \vdots \\ 0 & & 1 & -\lambda \end{pmatrix} \in \mathbb{R}^{n \times n}$$

$$\det(A - \lambda I) = (-\lambda)^n - \alpha(-1)^{n-2} = (-1)^n \lambda^n + (-1)^{n-1} \alpha = (-1)^n (\lambda^n - \alpha)$$

Ponendo l'espressione uguale a zero si ottiene il polinomio caratteristico. In questo caso, avremo radici reali solo per $\alpha > 0$ con:

$$\begin{cases} \lambda = \pm \sqrt[n]{\alpha} \text{ con } n \text{ pari} \\ \lambda = \sqrt[n]{\alpha} \text{ con } n \text{ dispari} \end{cases}$$

Esercizio 6.12

Dimostrare che i metodi di Jacobi e Gauss – Seidel possono essere utilizzati per la risoluzione del sistema lineare (gli elementi non indicati sono da intendersi nulli)

$$\begin{pmatrix} 1 & & & -\frac{1}{2} \\ -1 & 1 & & \\ & \ddots & \ddots & \\ & & -1 & 1 \end{pmatrix} x = \begin{pmatrix} \frac{1}{2} \\ 0 \\ \vdots \\ 0 \end{pmatrix} \in \mathbb{R}^n,$$

la cui soluzione è $x = (1, \dots, 1)^T \in \mathbb{R}^n$. Confrontare il numero di iterazioni richieste dai due metodi per soddisfare lo stesso criterio di arresto (6.30), per valori crescenti di n e per tolleranze ε decrescenti. Riportare i risultati ottenuti in una tabella (n/ε).

Svolgimento

Affinché un metodo di risoluzione del sistema indotto da uno splitting del tipo $A = M - N$ sia convergente, bisogna dimostrare che: $A^{-1} \geq 0$ e lo splitting sia regolare. Sappiamo già che i metodi di Jacobi e Gauss-Seidel si basano su splitting regolari. Dimostriamo quindi che la matrice della traccia è una M-matrice, ovvero che la sua inversa è maggiore e uguale a 0. Abbiamo quindi:

$$A = \begin{pmatrix} 1 & & & -\frac{1}{2} \\ -1 & 1 & & \\ & \ddots & \ddots & \\ & & -1 & 1 \end{pmatrix}$$

Consideriamo lo splitting $A = \alpha I - B$ dove deve essere $B \geq 0$ e $\rho(B) < \alpha$. Scriviamo lo splitting:

$$A = 1 * I - \begin{pmatrix} 0 & & & \frac{1}{2} \\ 1 & 0 & & \\ & \ddots & \ddots & \\ & & 1 & 0 \end{pmatrix}$$

Osserviamo quindi che $\alpha = 1$ e $B \geq 0$. Dimostriamo ora che $\rho(B) < \alpha$, quindi calcoliamo le radici del polinomio caratteristico: $\det(B - \lambda I)$.

$$(-\lambda)^n - \frac{1}{2}(-1)^{n-2} = (-1)^n \lambda^n + \frac{1}{2}(-1)^{n-1} = (-1)^n \left(\lambda^n - \frac{1}{2} \right)$$

Otteniamo le seguenti radici:

$$\begin{cases} \lambda = \pm \sqrt[n]{\frac{1}{2}}, & n \text{ pari} \\ \lambda = \sqrt[n]{\frac{1}{2}}, & n \text{ dispari} \end{cases}$$

Osserviamo quindi che a prescindere da n si ha $\rho(B) = \sqrt[n]{\frac{1}{2}} < 1$, pertanto A è una M-matrice.

Possiamo quindi applicare i metodi di Jacobi e Gauss-Seidel.

Per quanto riguarda le implementazioni MATLAB, avendo analogamente all'esercizio 6.2 una matrice sparsa a dimensione variabile, si è pensato di ottimizzare lo spazio occupato introducendo algoritmi di compressione. Utilizziamo uno script che itera sull'aumento della dimensione della matrice diminuendo la tolleranza. Lo script costruisce matrice e vettore dei termini noti secondo le specifiche della traccia, calcola il fattore di condizionamento della matrice e la reale tolleranza da passare ai metodi di risoluzione. Richiama poi e richiama la *function* 6.2.4 per la compressione della matrice. Successivamente viene richiamata una *function* che, a seconda di un parametro in input, esegue il metodo di risoluzione di Jacobi o Gauss-Seidel (6.12.2) appositamente modificati per la compressione. In tale *function* viene effettuato il prodotto MATVEC per righe (6.12.3) e richiamata la *function* 6.12.4 (per Jacobi) o 6.12.5 (per Gauss-Seidel) tramite funzione anonima.

[Codice MATLAB 6.12.1](#) (script)

[Codice MATLAB 6.12.2](#) (Jacobi o Gauss-Seidel con compressione)

[Codice MATLAB 6.12.3](#) (MATVEC per righe)

[Codice MATLAB 6.12.4](#) (Risoluzione sistema diagonale con compressione)

[Codice MATLAB 6.12.5](#) (Risoluzione sistema triangolare inferiore a diagonale unitaria con compressione)

Riportiamo infine i risultati ottenuti per Jacobi e Gauss-Seidel in due tabelle

Jacobi:

$n \backslash \varepsilon$	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}	10^{-6}	10^{-7}	10^{-8}	10^{-9}	10^{-10}
3	21	33	42	51	63	72	81	93	102	111
6	48	72	90	108	132	150	168	192	210	228
9	81	108	144	171	198	234	261	288	324	351
12	108	156	192	228	276	312	348	396	432	468
15	150	195	240	300	345	390	450	495	540	600
18	180	234	306	360	414	486	540	594	666	720
21	210	294	357	420	504	567	630	693	777	840
24	240	336	408	480	576	648	720	816	888	960
27	297	378	459	567	648	729	837	918	999	1080
30	330	420	510	630	720	810	930	1020	1110	1230

Gauss-Seidel:

$n \backslash \varepsilon$	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}	10^{-6}	10^{-7}	10^{-8}	10^{-9}	10^{-10}
3	8	12	16	19	23	26	29	32	35	39
6	10	14	18	21	24	27	31	34	38	41
9	13	16	19	22	24	28	32	35	39	42
12	13	17	20	23	26	30	33	37	39	43
15	13	17	20	23	27	30	34	37	41	44
18	14	18	21	24	28	31	34	38	41	44
21	15	18	21	25	28	31	34	38	41	45
24	15	18	21	25	28	31	35	38	41	44
27	15	19	22	25	29	32	35	39	42	45
30	16	19	22	25	29	32	36	39	42	45

Possiamo quindi concludere che il metodo di Gauss-Seidel converge alla soluzione con meno iterazioni rispetto al metodo di Jacobi.

Esercizio complementare

Sia data la matrice $n \times n$, $A = (a_{i,j})$ con

- $a_{i,i} = 1$
- $a_{i,j} = -\frac{1}{n}$, se $i \neq j$

Stabilire che i metodi di Gauss-Seidel e Jacobi sono convergenti. Quindi, determinato il termine noto che corrisponde alla soluzione costante di costante valore 1, graficare l'errore ottenuto, rispetto al numero dell'iterazione (in norma infinito), partendo dalla soluzione approssimata nulla.

Graficare (*semilogy*) per i casi $n = 10, 50, 100$ per un numero di iterazioni pari a $10n$, comparando i due metodi tra loro.

Svolgimento

La matrice in questione è la seguente:

$$A = \begin{pmatrix} 1 & -\frac{1}{n} & \dots & -\frac{1}{n} \\ -\frac{1}{n} & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & -\frac{1}{n} \\ -\frac{1}{n} & \dots & -\frac{1}{n} & 1 \end{pmatrix}$$

Dobbiamo stabilire se i metodi di Gauss-Seidel e Jacobi convergono. Come visto in precedenza, sapendo che lo splitting di tali metodi è regolare, basta verificare che $A^{-1} > 0$, proprietà

garantita se A è una M-matrice. Andiamo quindi a verificarlo usando la seguente scomposizione: $A = \alpha I - B$.

$$A = 1 * I - \begin{pmatrix} 0 & \frac{1}{n} & \dots & \frac{1}{n} \\ \frac{1}{n} & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \frac{1}{n} \\ \frac{1}{n} & \dots & \frac{1}{n} & 0 \end{pmatrix}$$

Notiamo quindi che $B \geq 0$. Inoltre, sapendo che ogni tipo di norma costituisce una maggiorazione del raggio spettrale, e osservando che $\|B\|_{1,\infty} = (n-1)\frac{1}{n}$ e quindi $\|B\|_2 \leq (n-1)\frac{1}{n}$, si vede subito che $\rho(B) = (n-1)\frac{1}{n} = 1 - \frac{1}{n}$. Tale valore è sempre minore di $\alpha = 1$ per ogni $n > 1$. Pertanto A è una M-matrice, e ne consegue che è possibile applicare i metodi di Jacobi e Gauss-Seidel.

Determiniamo ora il valore del vettore dei termini noti secondo quanto indicato dalla traccia:

$$\begin{pmatrix} 1 & -\frac{1}{n} & \dots & -\frac{1}{n} \\ -\frac{1}{n} & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & -\frac{1}{n} \\ -\frac{1}{n} & \dots & -\frac{1}{n} & 1 \end{pmatrix} \begin{pmatrix} 1 \\ \vdots \\ \vdots \\ \vdots \\ 1 \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ \vdots \\ \vdots \\ b_n \end{pmatrix}$$

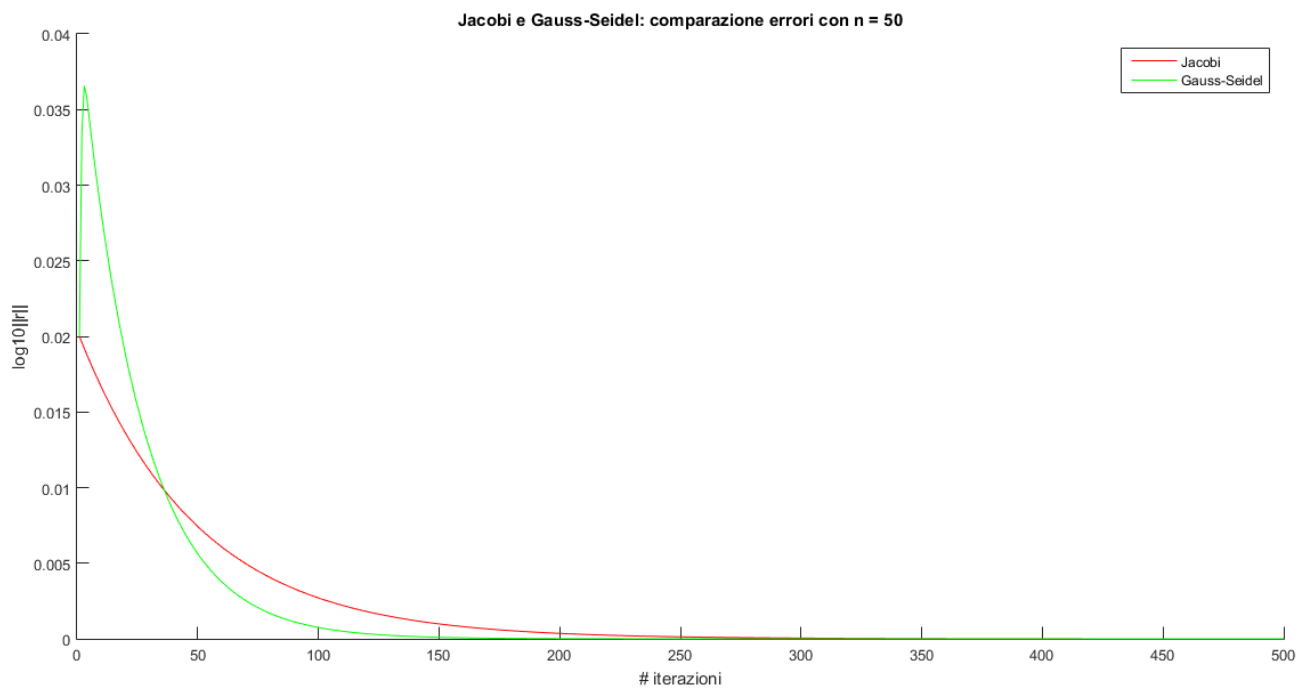
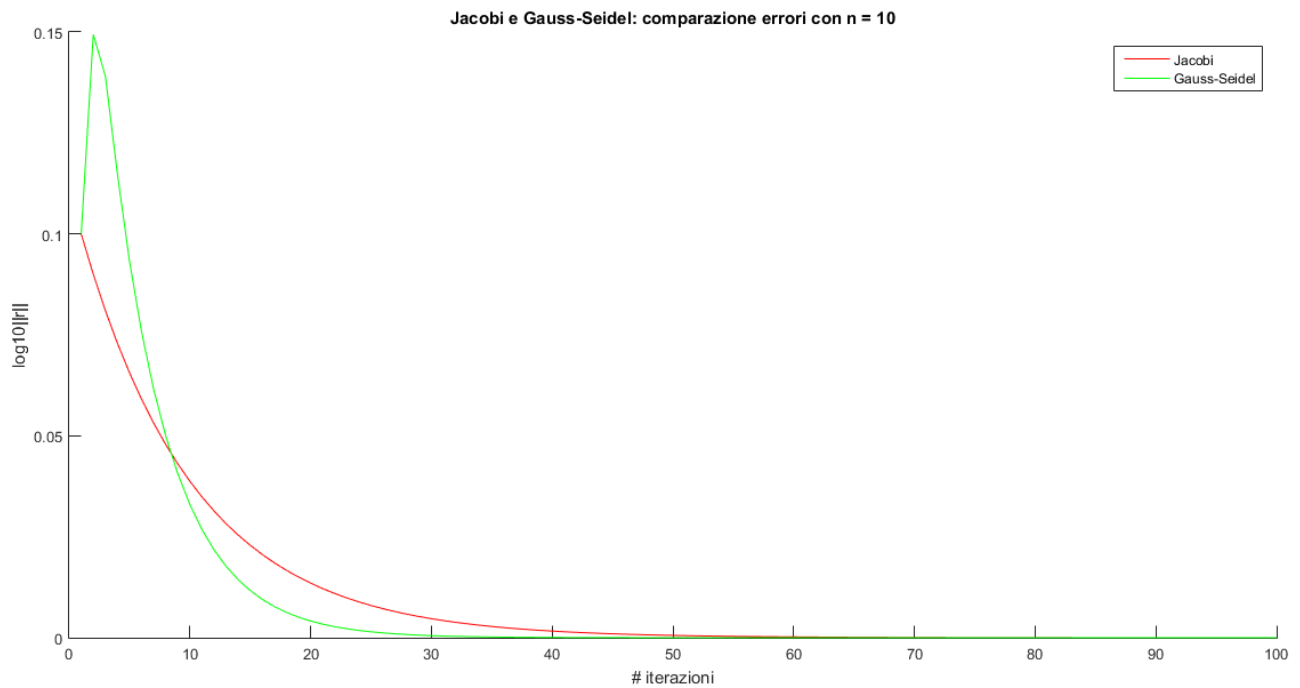
$$b_i = 1 - \left((n-1)\frac{1}{n} \right) = \frac{1}{n}, \quad \text{con } i = 1, \dots, n$$

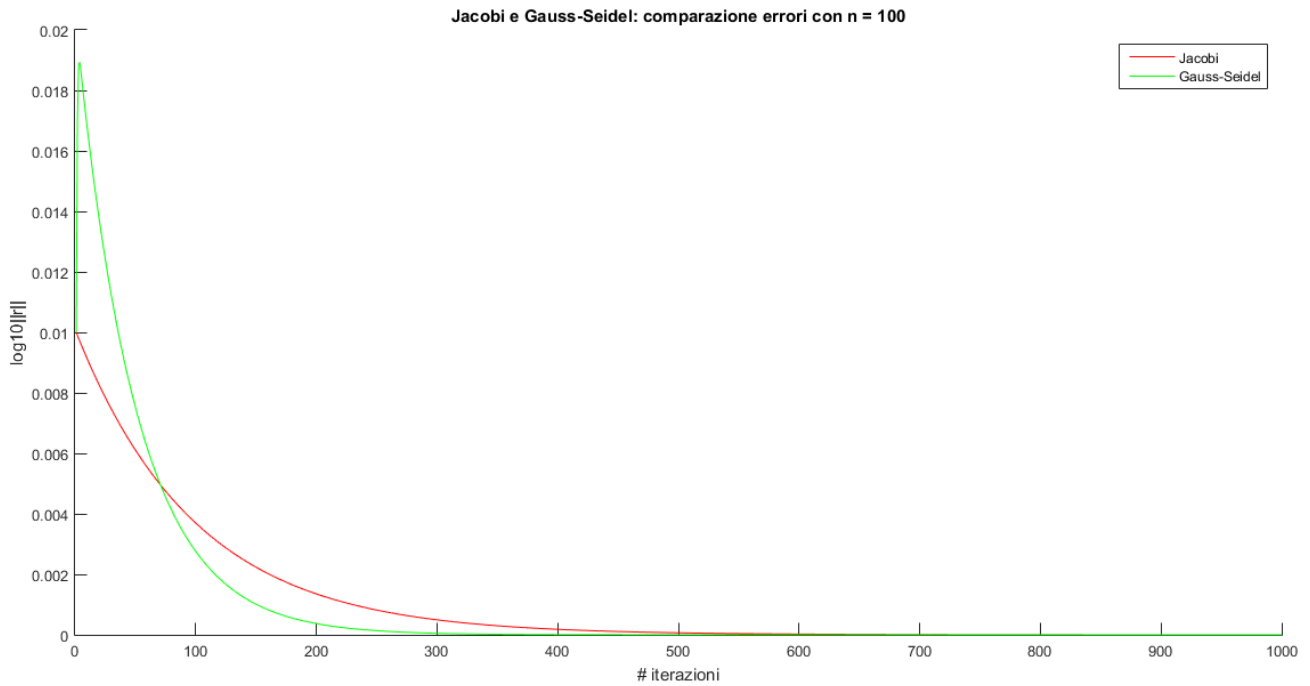
A questo punto siamo pronti per implementare il codice MATLAB. Presentiamo uno script che, stabilita un'arbitraria tolleranza di 10^{-5} , itera per 3 volte (tanti quanti sono i casi richiesti) e per ciascuna iterazione: costruisce l'opportuna matrice A e vettore b ; richiama una *function* dove a seconda di un parametro viene eseguito il metodo di Jacobi o Gauss-Seidel; crea infine il grafico di confronto richiesto. L'implementazione del metodo che presentiamo è ottimizzata in termini di occupazione di spazio (preallocazione degli elementi in memoria e riduzione dello spazio usato rispetto all'implementazione 'classica') e modificata opportunamente in funzione dell'esercizio: viene restituito anche un vettore contenente le norme infinito dei residui di ogni iterazione ed è stato aggiunto un criterio d'arresto ulteriore basato sul numero massimo $10n$ di iterazioni.

[Codice MATLAB 6.1 Complementare](#) (Script)

[Codice MATLAB 6.2 Complementare](#) (Jacobi o Gauss-Seidel)

Presentiamo infine i tre grafici di confronto:





Concludiamo osservando che, come ci si aspettava, la norma infinito del vettore residuo del metodo di Gauss-Seidel si appiattisce verso 0 prima di quello di Jacobi, ovvero il metodo di Gauss-Seidel converge alla soluzione in un numero di iterazioni minore rispetto a quello di Jacobi.

CODICI CAPITOLO 2

CODICE 2.1

```
% [ x ] = Ex_2_1(n, alpha, xi, iMax, tolX, g)
% Questa funzione definisce una procedura iterativa basata sul metodo di
% Newton per determinare la radice n-esima di alpha, con alpha > 0.
% E' possibile abilitare la parte grafica, che mostra correttamente
% l'andamento del metodo, solo quando la radice è pari a radice terza di 10.
% La versione del criterio d'arresto scelta e' quella che prevede anche
% l'utilizzo di una rtolX, così che il controllo diventa il 2.15 di
% pag. 35 del libro di testo.
% Input:
% - n: indice della radice (deve essere diverso da 0)
% - alpha: radicando (deve essere maggiore di 0)
% - xi: punto d'innescò (deve essere diverso da 0)
% - iMax: numero massimo di iterazioni prefissate
% - tolX: tolleranza iniziale usata per il calcolo della soglia d'arresto
% - g: se true abilita la parte grafica, false altrimenti
% Output:
% - x: approssimazione della radice
function [ x ] = Ex_2_1( n, alpha, xi, iMax, tolX, g )
fprintf('\n*** Metodo di Newton ***\n\n- Funzione: f(x) = x^n - alpha\n-
Criterio d'arresto: criterio dell'incremento relativo\n- TolX = %d\n- Punto
d'innescò: x0 = %d\n', tolX, xi);
if alpha <= 0
    error('alpha deve essere > 0');
end
if n == 0
    error('n deve essere diverso da 0');
end
if xi == 0
    error('Impossibile applicare il metodo di Newton con punto d'innescò x = 0
per questa specifica funzione poiché' la retta tangente in quel punto e' un
asintoto orizzontale di equazione y = -10');
end

if g
    if (alpha == 10 && n == 3)
        h = 10^(1/3);
        f = @(x)x.^n - alpha;
        fig = figure(1);
        set (fig, 'Units', 'normalized', 'Position', [0,0,1,1]);
        hold on;
        title('Metodo di Newton - Ex. 2.1');
        if xi < h
            fplot(f, [xi - h, h + 1], 'b');
        else
            fplot(f, [-1, xi + 1], 'b');
        end
        line([0 0], ylim, 'Color', 'black');
        line(xlim, [0 0], 'Color', 'black');
        leg(1) = plot(h, 0, 'g*');
        leg(2) = plot(xi, 0, 'r*');
        legend(leg, 'radice = 10^{1/3}', ['innescò = ', num2str(xi)]);
        pause;
    else
        fprintf('\nParte grafica disabilitata\n');
        g = 0;
    end
end
```

```

    end
end
tic;
i = 1;

while i < iMax
    x = 1/n*((n - 1)*xi + alpha/xi^(n - 1));
    % x = xi + (xi^(1-n)*(alpha - xi^n))/n; % questa fa meno operazioni
    fprintf('\nx%d = %.16f' , i, x);
    errOnX = abs(x - xi)/(1 + abs(x));
    if g
        fx = feval(f,xi);
        shg;
        plot([xi, xi],[fx, 0], 'r');
        pause;
        shg;
        plot([xi, x],[fx, 0], 'g');
        pause;
    end
    if (errOnX <= tolx)
        fprintf('\n\nIl metodo converge a %.16f\nNumero di iterazioni: %d\nTempo
di esecuzione: %d secondi\n\n', x, i, toc);
        return;
    end
    xi = x;
    i = i + 1;
end
fprintf('\n\nIl metodo NON converge!\nNumero di iterazioni: %d\nTempo di
esecuzione: %d secondi\n\n', i, toc);
end

```

CODICE 2.2

```

% [ x ] = Ex_2_2( n, alpha, xi_1, iMax, tolx, g)
% Questa funzione definisce una procedura iterativa basata sul metodo
% delle Secanti per determinare la radice n-esima di alpha, con alpha > 0.
% Il primo passo viene eseguito col metodo di Newton.
% La funzione all'inizio richiama Ex_2_2_ScomponiDifferenzaPotenze(n)
% per la composizione della procedura iterativa usata per approssimare la
% radice.
% E' possibile abilitare la parte grafica, che mostra correttamente
% l'andamento del metodo, solo quando la radice è pari a radice terza di 10.
% La versione del criterio d'arresto scelta è quella che prevede anche
% l'utilizzo di una rtolx, così che il controllo diventa il 2.15 di
% pag. 35 del libro di testo.
% Input:
% - n: indice della radice (deve essere diverso da 0)
% - alpha: radicando (deve essere maggiore di 0)
% - xi_1: punto d'innescio e approssimazione al passo (i-1)-esimo
%       (deve essere diverso da 0)
% - iMax: numero massimo di iterazioni prefissate
% - tolx: tolleranza iniziale usata per il calcolo della soglia d'arresto
% - g: se true abilita la parte grafica, false altrimenti
% Output:
% - x: approssimazione della radice
function [ x ] = Ex_2_2( n, alpha, xi_1, iMax, tolx, g)
fprintf('\n*** Metodo delle Secanti ***\n\n- Funzione: f(x) = x^n - alpha\n-
Criterio d'arresto: criterio dell''incremento relativo\n- Tolx = %d\n- Punto
d'innescio: x0 = %d\n\n', tolx, xi_1);
if (xi_1 == 0 || n == 0 || alpha <= 0)
    error('Impossibile applicare il metodo delle Secanti');
end

```

```

end
num = Ex_2_2_ScomponiDifferenzaPotenze(n - 1);
den = Ex_2_2_ScomponiDifferenzaPotenze(n);
syms a;
syms b;
syms c;
f = matlabFunction(simplify(((num*a*b) + c*(a - b))/den));

tic;
x = xi_1 + (xi_1^(1-n)*(alpha - xi_1^n))/n;
fprintf('x1 = %d <--- primo passo calcolato col metodo di Newton\n',x);
i = 1;

if (g)
    if (alpha == 10 && n == 3)
        h = 10^(1/3);
        fsec = @(x)x.^n - alpha;
        fig = figure(1);
        set (fig, 'Units', 'normalized', 'Position', [0,0,1,1]);
        hold on;
        title('Metodo delle Secanti - Ex. 2.2');
        if x < h
            fplot(fsec, [x - h, h + 1], 'b');
        else
            fplot(fsec, [-1, x + 1], 'b');
        end
        plot(h, 0, 'g*');
        line(xlim, [0 0], 'Color', 'black');
        legend('f(x) = x^{3} - 10', ['radice = ', num2str(h)]);
        pause;
    else
        fprintf('\nParte grafica disabilitata\n');
        g = 0;
    end
end
while ( i < iMax )
    xi = x;
    x = f(xi, xi_1, alpha);
    fprintf('x%d = %.16f\n', i, x);
    if g
        plot([xi, xi],[fsec(xi), 0], 'r');
        if (i > 1)
            plot([xi_1, x], [fsec(xi_1), 0], 'g');
        end
        plot([xi, x], [fsec(xi), fsec(x)], 'g');
        shg;
        pause;
    end
    errOnX = abs(x - xi) / (1 + abs(x));
    if (errOnX <= tolX)
        fprintf('\nIl metodo converge a %d\nNumero di iterazioni: %d\nTempo di
esecuzione: %d secondi\n\n', x, i, toc);
        return;
    end
    xi_1 = xi;
    i = i + 1;
end
fprintf('\nIl metodo NON converge!\nNumero di iterazioni: %d\nTempo di
esecuzione: %d secondi\n\n', i, toc);
end

```

CODICE 2.2.1

```
% [ f ] = Ex_2_2_ScomponiDifferenzaPotenze( n )
% Algoritmo per la scomposizione della differenza di due potenze con lo
% stesso esponente. Restituisce un function handler dove vengono usate
% le variabili simboliche 'a' e 'b'.
% Input:
% - n: grado del binomio
% Output:
% - f: function handler del binomio scomposto
function [ f ] = Ex_2_2_ScomponiDifferenzaPotenze( n )
syms a;
syms b;
f = 0;
for i = 1 : n
    f = f + a.^(n - i).*b.^(i - 1);
end
f = matlabFunction((a - b).*f);
end
```

CODICE 2.3.1

```
% [ x ] = Ex_2_3_1(f, df, x0, iMax, tolX, m, ac, g)
% Implementazione del metodo iterativo di Newton per il calcolo della
% radice di una funzione.
% A seconda dell'input dell'utente, il metodo potrà usare una costante
% asintotica (convergenza solo lineare) o ripristinare la convergenza
% quadratica propria del metodo usando un fattore m molteplicità
% algebrica della radice, se nota. Non ha senso usare la costante
% asintotica se è nota la molteplicità non semplice della radice.
% La versione del criterio d'arresto scelta è quella che prevede anche
% l'utilizzo di una rtolX, così che il controllo diventa il 2.15 di
% pag. 35 del libro di testo.
% Input:
% - f: la funzione
% - df: la derivata della funzione
% - x0: punto d'innescio
% - iMax: numero massimo di iterazioni prefissate
% - tolX: tolleranza iniziale usata per il calcolo della soglia d'arresto
% - m: molteplicità algebrica della radice. Se ignota, mettere 1.
% - ac: se true stima e usa la costante asintotica, false altrimenti
% - g: se true abilita la parte grafica, false altrimenti
% Output:
% - x: approssimazione della radice
function [ x ] = Ex_2_3_1(f, df, x0, iMax, tolX, m, ac, g)
if (m == 1 && ac)
    fprintf('\n*** Metodo di Newton ***\n\n- Criterio d'arresto: incremento
relativo con costante asintotica\n- Punto d'innescio: x0 = %d\n- TolX = %d\n',
x0, tolX);
    string = 'Metodo di Newton - Radice semplice con costante asintotica';
elseif(m == 1 && ~ac)
    fprintf('\n*** Metodo di Newton ***\n\n- Criterio d'arresto: incremento
relativo\n- Punto d'innescio: x0 = %d\n- TolX = %d\n', x0, tolX);
    string = 'Metodo di Newton - Radice semplice';
elseif (m ~= 1 && ac)
    fprintf('\n*** Metodo di Newton Modificato ***\n\n- Criterio d'arresto:
incremento relativo con costante asintotica\n(Attenzione: si sta usando la
costante asintotica per un metodo che si supponga convergere quadraticamente)\n-
Punto d'innescio: x0 = %d\n- TolX = %d\n- Molteplicità radice = %d\n', x0,
tolX, m);
```

```

        string = 'Metodo di Newton - Radice multipla con costante asintotica';
elseif (m ~= 1 && ~ac)
    fprintf('\n*** Metodo di Newton Modificato ***\n\n- Criterio d''arresto:
    incremento relativo\n- Punto d''innesco: x0 = %d\n- Tolx = %d\n- Molteplicita''
    radice = %d\n', x0, tolx, m);
    string = 'Metodo di Newton - Radice multipla';
end

if (g)
    fig = figure(1);
    set (fig, 'Units', 'normalized', 'Position', [0,0,1,1]);
    hold on;
    title(string);
    fplot(f, [-abs(x0) - 1, abs(x0) + 1], 'b');
    line(xlim, [0 0], 'Color', 'black');
    pause;
end

tic;

flag = false;           % Flag di controllo sulla costante asintotica
nVal = 0;               % Contatore di valutazioni di funzioni
i = 1;

if (ac)
    flag = true;
end

while ( i < iMax )

    if (flag)
        fx = feval(f, x0);
        dfx = feval(df, x0);
        nVal = 2;
        if (fx == 0)
            x = x0;
            fprintf('\n\nIl metodo converge a %1.16f\nNumero di iterazioni:
1\nNumero di valutazioni di funzioni: %d\nTempo di esecuzione: %d secondi\n\n',
x, nVal, toc);
            return
        end

        if (dfx == 0)
            x = x0;
            fprintf('\n\nDerivata prima uguale a zero --> impossibile continuare
l''iterazione\nApprossimazione della radice ottenuta: %d\nNumero di iterazioni:
%d\nNumero di valutazioni di funzioni: %d\nTempo di esecuzione: %d secondi\n\n',
x, i, nVal, toc);
            return;
        end

        x1 = x0 - m * (fx / dfx);
        fprintf('\nx1 = %d', x1);

        if (g)
            plot([x0 x1], [fx 0], 'g');
            plot([x0 x0], [0 fx], 'r');
            pause;
        end

        errOnX = abs(x1 - x0)/(1 + abs(x1));
    end
end

```

```

        if (errOnX <= tolX)
            x = x1;
            fprintf('\n\nIl metodo converge a %1.16f\nNumero di iterazioni:
1\nNumero di valutazioni di funzioni: %d\nTempo di esecuzione: %d secondi\n\n',
x, nVal, toc);
            return;
        end

        fx = feval(f, x1);
        dfx = feval(df, x1);
        nVal = nVal + 2;

        if (dfx == 0)
            x = x1;
            fprintf('\n\nDerivata prima uguale a zero --> impossibile continuare
1''iterazione\nApprossimazione della radice ottenuta: %1.16f\nNumero di
iterazioni: %d\nNumero di valutazioni di funzioni: %d\nTempo di esecuzione: %d
secondi\n\n', x, i, nVal, toc);
            return;
        end

        x2 = x1 - m * (fx / dfx);
        fprintf('\nx2 = %d', x2);

        if (g)
            plot([x1 x2], [fx 0], 'g');
            plot([x1 x1], [0 fx], 'r');
            pause;
        end

        c = abs(x2 - x1)/abs(x1 - x0);

        errOnX = abs(x2 - x1)/(1 + abs(x2));
        if (errOnX <= tolX * (1 - c)/c)
            x = x2;
            fprintf('\n\nIl metodo converge a %1.16f\nNumero di iterazioni:
2\nNumero di valutazioni di funzioni: %d\nCostante asintotica: %1.16f\nTempo di
esecuzione: %d secondi\n\n', x, nVal, c, toc);
            return;
        end
        x0 = x2;
        flag = false;
        i = 3;
    end

    fx = feval(f, x0);
    dfx = feval(df, x0);
    nVal = nVal + 2;

    if (fx == 0)
        x = x0;
        fprintf('\n\nIl metodo converge a %1.16f\nNumero di iterazioni:
%d\nNumero di valutazioni di funzioni: %d\nTempo di esecuzione: %d secondi\n\n',
x, i - 1, nVal, toc);
        return
    end

    if (dfx == 0)
        x = x0;
        fprintf('\n\nDerivata prima uguale a zero --> impossibile continuare
1''iterazione\nApprossimazione della radice ottenuta: %1.16f\nNumero di

```

```

iterazioni: %d\nNumero di valutazioni di funzioni: %d\nTempo di esecuzione: %d
secondi\n\n', x, i, nVal, toc);
    return;
end

x = x0 - m * (fx / dfx);
fprintf('\nx%d = %d', i, x);

if (g)
    plot([x0 x], [fx 0], 'g');
    plot([x0 x0], [0 fx], 'r');
    pause;
end

% Aggiornamento condizione d'uscita sulla tolX
errOnX = abs(x - x0)/(1 + abs(x));
if (ac)
    c = abs(x - x0)/abs(x0 - x1);
    if (errOnX <= tolX * (1 - c)/c)
        fprintf('\n\nIl metodo converge a %1.16f\nNumero di iterazioni:
%d\nNumero di valutazioni di funzioni: %d\nCostante asintotica: %1.16f\nTempo di
esecuzione: %d secondi\n\n', x, i, nVal, c, toc);
        return;
    end
    x1 = x0;
elseif (errOnX <= tolX)
    fprintf('\n\nIl metodo converge a %1.16f\nNumero di iterazioni:
%d\nNumero di valutazioni di funzioni: %d\nTempo di esecuzione: %d secondi\n\n',
x, i, nVal, toc);
    return;
end

x0 = x;
i = i + 1;
end
fprintf('\n\nIl metodo NON converge!\nNumero di iterazioni: %d\nNumero di
valutazioni di funzioni: %d\nTempo di esecuzione: %d secondi\n\n', i, nVal,
toc);
end

```

CODICE 2.3.2

```

% [ x ] = Ex_2_3_2( f, df, x0, iMax, tolX, g)
% Implementazione del metodo iterativo di Newton con accelerazione di
% Aitken per il calcolo della radice di una funzione.
% La versione del criterio d'arresto scelta e' quella che prevede anche
% l'utilizzo di una rtolX, cosi' che il controllo diventa il 2.15 di
% pag. 35 del libro di testo.
% Input:
% - f: la funzione
% - df: la derivata della funzione
% - x0: punto d'innescio
% - iMax: numero massimo di iterazioni prefissate
% - tolX: tolleranza iniziale usata per il calcolo della soglia d'arresto
% - g: se true abilita la parte grafica, false altrimenti
% Output:
% - x: approssimazione della radice
function [ x ] = Ex_2_3_2( f, df, x0, iMax, tolX, g)

```

```

fprintf('\n*** Metodo di Newton con Accelerazione di Aitken ***\n\n- Criterio
d'arresto: incremento relativo\n- Tolx = %d\n- Punto d'innesco: x0 = %d\n',
tolx, x0);

conv = false; % Booleana di controllo convergenza

nVal = 0;          % Contatore di valutazioni di funzioni
i = 0;            % Inizializzazione indice

if (g)
    fig = figure(1);
    set (fig, 'Units', 'normalized', 'Position', [0,0,1,1]);
    hold on;
    title('Metodo di Accelerazione di Aitken');
    fplot(f, [- abs(x0) - 1, abs(x0) + 1], 'b');
    line(xlim, [0 0], 'Color', 'black');
    pause;
end
tic;
x = x0;
while (i <= iMax)
    i = i + 1;
    x0 = x; % Aggiornamento della x

    fx = feval(f, x0);      % Valutazione funzione e derivata in x
    dfx = feval(df, x0);    % per il primo Newton intermedio
    nVal = nVal + 2;

    if (fx == 0)
        conv = true;
        break;
    end

    if (dfx == 0)
        fprintf('\n\nPrimo passo di Newton interno: derivata prima uguale a zero
--> impossibile continuare l'iterazione\nApprossimazione della radice ottenuta:
%2.16f\nNumero di iterazioni: %d\nNumero di valutazioni di funzioni: %d\nTempo
di esecuzione: %d secondi\n\n', x, i, nVal, toc);
        return;
    end

    x1 = x0 - fx/dfx;      % Calcolo x del primo Newton intermedio

    if g
        plot([x0 x1], [fx 0], 'r');
        plot([x0 x0], [0 fx], 'black');
        pause;
    end

    fx = feval(f, x1);      % Valutazione funzione e derivata in x
    dfx = feval(df, x1);    % per il secondo Newton intermedio
    nVal = nVal + 2;

    if (fx == 0)
        conv = true;
        x = x1;
        break;
    end

    if (dfx == 0)

```



```

        fprintf('\n\nSecondo passo di Newton interno: derivata prima uguale a
zero --> impossibile continuare l''iterazione\nApprossimazione della radice
ottenuta: %2.16f\nNumero di iterazioni: %d\nNumero di valutazioni di funzioni:
%d\nTempo di esecuzione: %d secondi\n\n', x, i, nVal, toc);
        return;
    end

    x2 = x1 - fx / dfx;          % Calcolo x del secondo passo di Newton

    if g
        plot([x1 x2], [fx 0], 'r');
        plot([x1 x1], [0 fx], 'black');
        pause;
    end

    den = x2 - 2*x1 + x0;        % Calcolo denominatore funzione accelerazione
    if (den == 0)                % di Aitken per evitare cancellazione numerica
        % Esponiamo il valore verosimilmente piu' accurato:
        x = x2;                 % il risultato dell'ultimo passo di Newton
        % interno
        fprintf('\n\nPasso esterno di Aitken: denominatore uguale a zero -->
impossibile continuare l''iterazione.\nApprossimazione della radice ottenuta:
%2.16f\nNumero di iterazioni: %d\nNumero di valutazioni di funzioni: %d\nTempo
di esecuzione: %d secondi\n\n', x, i, nVal, toc);
        return;
    end

    % Calcolo x dell'Accelerazione di Aitken
    x = (x2 * x0 - x1^2)/den;
    fprintf('\nx%d = %d', i, x);

    if g
        fx = feval(f, x2);
        plot([x2 x], [fx 0], 'g');
        plot([x2 x2], [0 fx], 'black');
        pause;
    end

    % Aggiornamento condizione d'uscita
    errOnX = abs(x - x0) / (1 + abs(x));
    if (errOnX <= tolX)
        conv = true;
        break;
    end
end

if (conv)
    fprintf('\n\nIl metodo converge a %2.16f\nNumero di iterazioni: %d\nNumero
di valutazioni di funzioni: %d\nTempo di esecuzione: %d secondi\n\n', x, i,
nVal, toc);
else
    fprintf('\n\nIl metodo NON converge!\nNumero di iterazioni: %d\nNumero di
valutazioni di funzioni: %d\nTempo di esecuzione: %d secondi\n\n', i , nVal,
toc);
end
end

```

CODICE 2.6.1

```
% [ x ] = Ex_2_6_1(f, df, x0, iMax, tolX, g)
% Implementazione del metodo iterativo delle secanti per il calcolo della
% radice di una funzione. L'algoritmo esegue un passo fuori dal ciclo
% principale in modo da poter applicare la formula delle secanti
% all'interno del ciclo. Il primo passo utilizza la formula di Newton.
% Viene controllato all'interno del ciclo il denominatore della formula
% per garantire la robustezza dell'algoritmo.
% La versione del criterio d'arresto scelta e' quella che prevede anche
% l'utilizzo di una rtolX, cosi' che il controllo diventa il 2.15 di
% pag. 35 del libro di testo.
% Input:
%   - f: la funzione
%   - df: la derivata prima della funzione
%   - x0: approssimazione iniziale della x
%   - iMax: numero massimo di iterazioni prefissate
%   - tolX: tolleranza desiderata
%   - g: true per abilitare la parte grafica, false altrimenti
% Output:
%   - x: approssimazione della radice
function [ x ] = Ex_2_6_1( f, df, x0, iMax, tolX, g)
fprintf('\n*** Metodo delle Secanti ***\n\n Criterio d'arresto: incremento
relativo\n- Punto d'ingresso: x0 = %d\n- TolX = %d\n', x0, tolX);

if (g)
    hax = axes;
    hold on;
    fplot(f, [0, 3], 'b');
    fplot(@(x) 0, get(hax, 'XLim'), 'black');
    title('Metodo delle Secanti');
    pause;
end
tic;
fx = feval(f, x0);
dfx = feval(df, x0);
nVal = 2;

if (fx == 0)
    x = x0;
    fprintf('\n\nIl metodo converge a %.16f\nNumero di iterazioni: 0\nNumero di
valutazioni di funzioni: %d\nTempo di esecuzione: %d secondi\n\n', x, nVal,
toc);
    return;
end

x = x0 - fx / dfx;
% fprintf('x1 = %d    <--- primo passo calcolato col metodo di Newton', x);

if (g)
    plot([x0 x], [fx 0], 'r');
    plot([x0 x0], [0 fx], 'black');
    pause;
end

errOnX = abs(x - x0) / (1 + abs(x));
if (errOnX <= tolX)
    fprintf('\n\nIl metodo converge a %.16f\nNumero di iterazioni: 1\nNumero di
valutazioni di funzioni: %d\nTempo di esecuzione: %d secondi\n\n', x, nVal,
toc);
    return;
```

```

end

i = 2;
while ( i < iMax )
    fx0 = fx;
    fx = feval(f, x);
    nVal = nVal + 1;

    if (fx == 0)
        fprintf('\n\nIl metodo converge a %2.16f\nNumero di iterazioni:
%d\nNumero di valutazioni di funzioni: %d\nTempo di esecuzione: %d secondi\n\n',
x, i, nVal, toc);
        return;
    end

    den = fx - fx0;

    if (den == 0)
        error('Denominatore uguale a zero --> impossibile applicare il metodo
delle Secanti');
    end

    x1 = (fx*x0 - fx0*x) / (fx - fx0);
    %     fprintf('\nx%d = %d', i, x1);

    if (g)
        plot([x x1], [fx 0], 'r');
        plot([x x], [0 fx], 'black');
        pause;
    end

    errOnX = abs(x1 - x) / (1 + abs(x1));
    if (errOnX <= tolX)
        x = x1;
        fprintf('\n\nIl metodo converge a %2.16f\nNumero di iterazioni:
%d\nNumero di valutazioni di funzioni: %d\nTempo di esecuzione: %d secondi\n\n',
x, i, nVal, toc);
        return;
    end
    x0 = x;
    x = x1;
    i = i + 1;
end
fprintf('\n\nIl metodo NON converge!\nNumero di iterazioni: %d\nNumero di
valutazioni di funzioni: %d\nTempo di esecuzione: %d secondi\n\n', i , nVal,
toc);
end

```

CODICE 2.6.2

```

% [ x ] = Ex_2_6_2( f, df, x0, iMax, tolX, ac, g)
% Implementazione del metodo iterativo delle corde per il calcolo della
% radice di una funzione. L'algoritmo esegue due passi fuori dal ciclo
% principale in modo da poter eventualmente calcolare la costante asintotica.
% Il primo passo utilizza la formula di Newton, il secondo quella delle corde.
% La versione del criterio d'arresto scelta e' quella che prevede anche
% l'utilizzo di una rtolX, cosi' che il controllo diventa il 2.15 di
% pag. 35 del libro di testo.
% Input:
% - f: la funzione

```

```

% - df: la derivata prima della funzione
% - x0: approssimazione iniziale della x
% - iMax: numero massimo di iterazioni prefissate
% - tolX: tolleranza desiderata
% - ac: booleana sulla costante asintotica: 1 la calcola e usa, 0 altrimenti
% - g: true per abilitare la parte grafica, false altrimenti
% Output:
% - x: approssimazione della radice
function [ x ] = Ex_2_6_2( f, df, x0, iMax, tolX, ac, g)

if (ac)
    fprintf('\n*** Metodo delle Corde ***\n\n Criterio d\'arresto: incremento
relativo con uso della costante asintotica\n- TolX = %d\n- Punto d\'innesco: x0
= %d\n', tolX, x0);
else
    fprintf('\n*** Metodo delle Corde ***\n\n Criterio d\'arresto: incremento
relativo\n- TolX = %d\n- Punto d\'innesco: x0 = %d\n', tolX, x0);
end

if (g)
    hax = axes;
    hold on;
    fplot(f, [-1, 3], 'b');
    fplot(@ (x) 0, get(hax, 'XLim'), 'black');
    title('Metodo delle Corde');
    pause
end
tic;
fx = feval(f, x0);
dfx = feval(df, x0);
nVal = 2;

if (fx == 0)
    x = x0;
    fprintf('\n\nIl metodo converge a %2.16f\nNumero di iterazioni: 0\nNumero di
valutazioni di funzioni: %d\nTempo di esecuzione: %d secondi\n\n', x, nVal,
toc);
    return;
end

if (dfx == 0)
    error('Derivata prima uguale a zero --> impossibile applicare il primo passo
(Newton) del metodo delle Corde');
end

x1 = x0 - fx / dfx;
% fprintf('\nx1 = %d <--- primo passo con Newton', x1);

if (g)
    plot([x0 x1], [fx 0], 'r');
    plot([x0 x0], [0 fx], 'black');
    pause;
end

% Aggiornamento condizione d\'uscita sulla tolX
errOnX = abs(x1 - x0) / (1 + abs(x1));
if (errOnX <= tolX)
    x = x1;
    fprintf('\n\nIl metodo converge a %2.16f\nNumero di iterazioni: 1\nNumero di
valutazioni di funzioni: %d\nTempo di esecuzione: %d secondi\n\n', x, nVal,
toc);

```

```

        return;
    end

    fx = feval(f, x1);
    nVal = nVal + 1;

    x2 = x1 - fx / dfx;
    % fprintf('\nx2 = %d', x2);

    if (g)
        plot([x1 x2], [fx 0], 'r');
        plot([x1 x1], [0 fx], 'black');
        pause;
    end

    errOnX = abs(x2 - x1) / (1 + abs(x2));

    if (ac)
        c = abs(x2 - x1)/abs(x1 - x0);
        if (errOnX <= tolX * (1 - c)/c)
            x = x2;
            fprintf('\n\nIl metodo converge a %2.16f\nNumero di iterazioni:
2\nNumero di valutazioni di funzioni: %d\nCostante asintotica: %d\nTempo di
esecuzione: %d secondi\n\n', x, nVal, c, toc);
            return;
        end
    elseif (errOnX <= tolX)
        x = x2;
        fprintf('\n\nIl metodo converge a %2.16f\nNumero di iterazioni: 2\nNumero di
valutazioni di funzioni: %d\nTempo di esecuzione: %d secondi\n\n', x, nVal,
toc);
        return;
    end

    i = 3;

    while (i < iMax)
        fx = feval(f, x2);
        nVal = nVal + 1;
        x = x2 - fx / dfx;
        % fprintf('\nx%d = %d', i, x);

        if (g)
            plot([x2 x], [fx 0], 'r');
            plot([x2 x2], [0 fx], 'black');
            pause;
        end

        errOnX = abs(x - x2) / (1 + abs(x));
        if (ac)
            c = abs(x - x2)/abs(x2 - x1);
            if (errOnX <= tolX * (1 - c)/c)
                fprintf('\n\nIl metodo converge a %2.16f\nNumero di iterazioni:
%d\nNumero di valutazioni di funzioni: %d\nCostante asintotica: %d\nTempo di
esecuzione: %d secondi\n\n', x, i, nVal, c, toc);
                return;
            end
            x1 = x2;
        elseif (errOnX <= tolX)

```

```

        fprintf('\n\nIl metodo converge a %2.16f\nNumero di iterazioni:
%d\nNumero di valutazioni di funzioni: %d\nTempo di esecuzione: %d secondi\n\n',
x, i, nVal, toc);
        return
    end
    x2 = x;
    i = i + 1;
end
fprintf('\n\nIl metodo NON converge!\nNumero di iterazioni: %d\nNumero di
valutazioni di funzioni: %d\nTempo di esecuzione: %d secondi\n\n', i, nVal,
toc);
end

```

CODICE 2.8

```

% [ x ] = Ex_2_8( f, a, b, tolX, g)
% Implementazione del metodo iterativo delle bisezione per il calcolo della
% radice di una funzione. La versione del criterio d'arresto scelta e'
% quella che prevede anche l'utilizzo di una rtolX, cosi' che il controllo
% diventa il 2.15 di pag. 35 del libro di testo.
% Input:
% - f: la funzione
% - a: primo estremo dell'intervallo di confidenza
% - b: secondo estremo dell'intervallo di confidenza
% - tolX: tolleranza desiderata
% - g: true per abilitare la parte grafica, false altrimenti
% Output:
% - x: approssimazione della radice
function [ x ] = Ex_2_8( f, a, b, tolX, g)
fprintf('\n*** Metodo di Bisezione ***\n\n- Criterio d'arresto: incremento
relativo\n- TolX = %d\n- Intervallo iniziale: [%d, %d]\n', tolX, a, b);

if (g)
    hax = axes;
    hold on;
    fplot(f, [a - 0.5, b + 0.5], 'b');
    fplot(@(x) 0, get(hax, 'XLim'), 'black');
    title('Metodo di Bisezione');
    line([a a], get(hax, 'YLim'), 'Color', [0 0 0])
    line([b b], get(hax, 'YLim'), 'Color', [0 0 0])
    pause;
end
tic;
conv = false;
imax = ceil(log2(b - a) - log2(tolX));
fa = feval(f,a);
fb = feval(f,b);
nVal = 2;

for i = 0 : imax
    x = (a + b) / 2;
    % fprintf('\nx%d = %d', i, x);
    fx = feval(f, x);
    nVal = nVal + 1;
    flx = abs((fb - fa) / (b - a));
    if abs(fx) <= tolX * flx
        conv = true;
        break;
    elseif fa * fx < 0
        if (g)
            YLim=get(hax,'YLim');

```

```

        rectangle('Position', [x YLim(1) abs(b-x) abs(YLim(2)-YLim(1))],
'FaceColor', [0 0 0]);
        pause;
    end
    b = x;
    fb = fx;
else
    if (g)
        YLim=get(hax,'YLim');
        rectangle('Position', [a YLim(1) abs(x-a) abs(YLim(2)-YLim(1))],
'FaceColor', [0 0 0]);
        pause;
    end
    a = x;
    fa = fx;
end
end
if (g)
    line([x x], get(hax, 'YLim'), 'Color', [1 0 0]);
    pause;
end
if (conv)
    fprintf('\n\nIl metodo converge a %.16f\nNumero di iterazioni: %d\nNumero
di valutazioni di funzioni: %d\nTempo di esecuzione: %d secondi\n\n', x, i,
nVal, toc);
else
    fprintf('\n\nIl metodo NON converge!\nNumero di iterazioni: %d\nNumero di
valutazioni di funzioni: %d\nTempo di esecuzione: %d secondi\n\n', i, nVal,
toc);
end
end
end

```

CODICI CAPITOLO 3

CODICE 3.1.1

```
% [ b ] = Ex_3_1_1(A, b)
% Questa funzione definisce una procedura basata sull'algoritmo 3.1 a pag.
% 46 del libro di testo per risolvere un sistema lineare triangolare inferiore
% a diagonale unitaria con opportune modifiche per la robustezza del codice.
% Il vettore b in input viene riscritto e restituito.
% Input:
% - A: matrice dei coefficienti
% - b: vettore dei termini noti
% Output:
% - b: vettore della soluzione del sistema
function[b] = Ex_3_1_1(A, b)

[m, n] = size(A);

if m~=n
    error('La matrice non è quadrata');
else
    for j = 1 : n
        if A(j,j) == 1
            for i = j + 1 : n
                b(i) = b(i) - A(i,j) * b(j);
            end
        else
            error('La matrice non è a diagonale unitaria');
        end
    end
end
end
```

CODICE 3.1.2

```
% [ b ] = Ex_3_1_2(A, b)
% Questa funzione definisce una procedura basata sull'algoritmo 3.1 a pag.
% 46 del libro di testo per risolvere un sistema lineare triangolare inferiore
% a diagonale unitaria. Il vettore b in input viene riscritto e restituito.
% Input:
% - A: matrice dei coefficienti
% - b: vettore dei termini noti
% Output:
% - b: vettore della soluzione del sistema
function[ b ] = Ex_3_1_2(A, b )
n = size(A,1);
for j = 1 : n
    for i = j + 1 : n
        b(i) = b(i) - A(i,j) * b(j);
    end
end
end
```

CODICE 3.2.1

```
% [ B ] = Ex_3_2_1(A, B)
% Questa funzione definisce una procedura che richiama l'algoritmo di
% fattorizzazione LU per fattorizzare una matrice A e risolve  $K \geq 1$ 
% sistemi lineari. La risoluzione di tali sistemi sfrutta algoritmi di
% risoluzione di sistemi triangolari.
% Input:
% - A: matrice dei coefficienti
% - B: matrice/vettore dei termini noti
% Output:
% - B: matrice/vettore contenente su ogni colonna il vettore soluzione
% dell' i-esimo sistema lineare con  $i = 1, \dots, k$ 
function [B] = Ex_3_2_1 (A, B)
A = Ex_3_2_2(A);
k = size(B, 2);
for i = 1 : k
    B(:, i) = Ex_3_1_2(A, B(:, i));
    B(:, i) = Ex_3_2_3(A, B(:, i));
end
end
```

CODICE 3.2.2

```
% [ A ] = Ex_3_2_2( A )
% Questa funzione definisce una procedura che presa in input una matrice,
% restituisce la matrice stessa fattorizzata LU.
% Input:
% - A: matrice dei coefficienti
% Output:
% - A: matrice fattorizzata LU
function [ A ] = Ex_3_2_2( A )
[m,n] = size(A);
if m~=n
    error('La matrice non è quadrata');
else
    for i = 1 : n - 1
        if A(i,i) == 0
            error('La matrice non è fattorizzabile LU');
        end
        A(i + 1:n, i) = A(i+1:n, i) / A(i,i);
        A(i+1:n, i+1:n) = A(i+1:n, i+1:n) - A(i+1 : n, i) * A(i, i+1:n);
    end
end
end
```

CODICE 3.2.3

```
% [ b ] = Ex_3_2_3(A, b)
% Questa funzione definisce una procedura basata sull'algoritmo 3.3 a pag.
% 47 del libro di testo per risolvere un sistema lineare triangolare
% superiore. Il vettore b in input viene riscritto e restituito.
% Input:
% - A: matrice dei coefficienti
% - b: vettore dei termini noti
% Output:
% - b: vettore della soluzione del sistema
```

```

function [ b ] = Ex_3_2_3(A, b)
[n] = size(A, 1);
for i = n : -1 : 1
    for j = i + 1 : n
        b(i) = b(i) - A(i,j) * b(j);
    end
    b(i) = b(i) / A(i,i);
end
end

```

CODICE 3.3.1

```

% [ B ] = Ex_3_3_1(A, B)
% Questa funzione definisce una procedura che richiama l'algoritmo di
% fattorizzazione LDT per fattorizzare A e risolve K >= 1 sistemi lineari.
% La risoluzione di tali sistemi sfrutta algoritmi di risoluzione
% di sistemi triangolari e di sistemi con matrici diagonali.
% Input:
% - A: matrice dei coefficienti
% - B: matrice/vettore dei termini noti
% Output:
% - B: matrice/vettore contenente su ogni colonna il vettore soluzione
% dell' i-esimo sistema lineare con i = 1,...,k
function [ B ] = Ex_3_3_1(A, B)
A = Ex_3_3_2(A);
k = size(B, 2);
for i = 1 : k
    B(:, i) = Ex_3_1_2(A, B(:, i));
    B(:, i) = Ex_3_3_3(A, B(:, i));
    B(:, i) = Ex_3_3_4(A', B(:, i));
end
end

```

CODICE 3.3.2

```

% [ A ] = Ex_3_3_2( A )
% Questa funzione definisce una procedura che presa in input una matrice,
% restituisce la matrice stessa fattorizzata LDL^T.
% Input:
% - A: matrice dei coefficienti
% Output:
% - A: matrice fattorizzata LDL^T
function [ A ] = Ex_3_3_2( A )
[m, n] = size(A);
if m ~= n
    error('La matrice non è quadrata');
end
if A(1,1) <= 0
    error('La matrice non è SDP');
end
A(2 : n, 1) = A(2 : n, 1) / A(1,1);
for j = 2 : n
    v = (A(j, 1 : j - 1).').*diag ( A (1 : j - 1, 1 : j - 1) );
    A(j, j) = A(j, j) - A(j, 1 : (j - 1)) * v;
    if A(j, j) <= 0
        error('La matrice non è SDP');
    end
    A(j + 1 : n, j) = (A(j + 1 : n, j) - A(j + 1 : n, 1 : j - 1)*v) / A(j, j);
end
end

```

```
end
```

CODICE 3.3.3

```
% [ b ] = Ex_3_3_3( A, b)
% Questa funzione definisce una procedura che risolve un sistema lineare
% in cui la matrice dei coefficienti è diagonale.
% Input:
% - A: matrice dei coefficienti quadrata
% - x: vettore dei termini noti.
% Output:
% - x: vettore soluzione del sistema
function [ b ] = Ex_3_3_3(A, b)
n = size(A);
for i = 1 : n
    b(i) = b(i) / A(i, i);
end
end
```

CODICE 3.3.4

```
% [ b ] = Ex_3_3_4(A, b)
% Questa funzione definisce una procedura basata sull'algoritmo 3.3 a pag.
% 47 del libro di testo per risolvere un sistema lineare triangolare superiore
% a diagonale unitaria. Il vettore b in input viene riscritto e restituito.
% Input:
% - A: matrice dei coefficienti
% - b: vettore dei termini noti
% Output:
% - b: vettore della soluzione del sistema
function [b] = Ex_3_3_4(A, b)
n = size(A,1);
for i = n : -1 : 1
    for j = i + 1 : n
        b(i) = b(i) - A(i,j) * b(j);
    end
end
end
end
```

CODICE 3.4.1

```
% [ B ] = Ex_3_4_1(A, B)
% Questa funzione definisce una procedura che richiama l'algoritmo di
% fattorizzazione LU con pivoting parziale per fattorizzare una matrice A
% e risolvere K >= 1 sistemi lineari.
% La risoluzione di tali sistemi sfrutta algoritmi di risoluzione
% di sistemi triangolari.
% Input:
% - A: matrice dei coefficienti
% - B: matrice/vettore dei termini noti
% Output:
% - B: matrice/vettore contenente su ogni colonna il vettore soluzione
% dell'i-esimo sistema lineare con i = 1,...,k
function [ B ] = Ex_3_4_1(A, B)
[A, p] = Ex_3_4_2(A);
k = size(B, 2);
for i = 1 : k
```

```

B(:, i) = Ex_3_1_2(A, B(p, i));
B(:, i) = Ex_3_2_3(A, B(:, i));
end
end

```

CODICE 3.4.2

```

% [ A, p ] = Ex_3_4_2( A )
% Questa funzione definisce una procedura che presa in input una matrice,
% restituisce la matrice stessa fattorizzata LU con pivoting parziale.
% Inoltre viene generato e restituito il vettore di permutazione p.
% Input:
% - A: matrice dei coefficienti
% Output:
% - A: matrice fattorizzata LU con pivoting parziale
% - p: vettore di permutazione
function [ A, p ] = Ex_3_4_2( A )
[m,n] = size(A);
if m~=n
    error('La matrice non è quadrata');
end
p = 1 : n;
for i = 1 : n - 1
    [mi, ki] = max(abs(A(i : n, i)));
    if mi == 0
        error('La matrice è singolare');
    end
    ki = ki + i - 1;
    if ki > i
        A([i,ki], :) = A([ki,i], :);
        p([i, ki]) = p([ki,i]);
    end
    A(i + 1 : n, i) = A(i + 1 : n, i) / A(i,i);
    A(i + 1 : n, i + 1 : n) = A(i + 1 : n, i + 1 : n) - A(i + 1 : n, i) * A(i, i
+ 1 : n);
end
end

```

CODICE 3.6

```

% [ x ] = Ex_3_6(F, J, x0, iMax, tol)
% Questa funzione definisce una procedura che applica il metodo di Newton
% a sistemi non lineari. Sfrutta la funzione definita nell'esercizio 3.4
% per la risoluzione di sistemi con pivoting parziale.
% Input:
% - F: matrice di funzioni
% - J: matrice Jacobiana
% - x0: punto d'innescio
% - iMax: massimo numero di iterazioni
% - tol: tolleranza desiderata
% Output:
% - x: approssimazione della radice
function [ x ] = Ex_3_6(F, J, x0, iMax, tol)
fx = -feval(F, x0);
jx = feval(J, x0);
solSistema = Ex_3_4_1(jx, fx);
x = x0 + solSistema;
i = 0;
while (norm(x - x0) > tol && i < iMax)
    x0 = x;

```

```

    fx = -feval(F, x0);
    jx = feval(J, x0);
    solSistema = Ex_3_4_1(jx, fx);
    x = x0 + solSistema;
    i = i + 1;
end
if norm(x - x0) > tol
    error('Il metodo non converge, usare un punto d''innesco piu'' vicino alle radici.');
```

CODICE 3.8.1

```

% [ B, r] = Ex_3_8_1(A, B)
% Questa funzione definisce una procedura che richiama l'algoritmo di
% fattorizzazione QR per fattorizzare una matrice A e risolvere K >= 1
% sistemi lineari. La risoluzione di tali sistemi sfrutta l'algoritmo di
% risoluzione di sistemi triangolari superiori.
% Input:
% - A: matrice dei coefficienti
% - B: matrice/vettore dei termini noti
% Output:
% - B: matrice/vettore contenente su ogni colonna il vettore soluzione
% dell' i-esimo sistema lineare con i = 1,...,k
function [ B, r] = Ex_3_8_1( A, B)
A = Ex_3_8_2(A);
[m,n] = size(A);
k = size(B, 2);
for j = 1 : k
    for t = 1 : n
        v = [1; A(t + 1 : m, t)];
        B( t : m, j ) = B( t : m, j ) - (( 2 * (v) * v' ) / ( v' * v ) ) * B( t :
m, j );
    end
    B(1 : n, j) = Ex_3_2_3(A (1 : n , 1 : n), B(1 : n, j));
end
r = norm(B(n + 1 : m));
B = B(1 : n, 1 : k);
end
```

CODICE 3.8.2

```

% [ A ] = Ex_3_8_2( A )
% Questa funzione definisce una procedura per fattorizzare QR una matrice
% A data in input. Restituisce la matrice stessa fattorizzata.
% Input:
% - A: matrice dei coefficienti
% Output:
% - A: matrice fattorizzata QR
function [ A ] = Ex_3_8_2( A )
[m,n] = size(A);
if n >= m
    error('Il numero di righe deve essere strettamente maggiore del numero delle colonne');
```

```

        error('La matrice non ha rango massimo');
    end
    if A(i,i) >= 0
        alpha = -alpha;
    end
    v1 = A(i,i) - alpha;
    A(i,i) = alpha;
    A(i + 1 : m, i) = A(i + 1 : m, i) / v1;
    beta = -v1 / alpha;
    A( i : m, i + 1 : n) = A( i : m, i + 1 : n) - (beta * [1; A(i + 1: m, i)]) *
    ([1 A(i + 1: m, i)'] * A(i : m, i + 1 : n));
end
end

```

CODICI CAPITOLO 4

CODICE 4.6

```
% [ f ] = Ex_4_6( x, f )
% Dato un set d'ascisse e relative valutazioni, calcola il vettore
% della differenza divisa.
% Input:
% -x: vettore contenente le ascisse sulle quali calcolare la differenza
%      divisa
% -f: vettore contenente i valori assunti dalla funzione in
%      corrispondenza delle ascisse x
% Output:
% -f: vettore contenente i valori delle differenze divise calcolate
function [ f ] = Ex_4_6( x, f )
n = length(x) - 1;
for j = 1 : n
    for i = n + 1 : -1 : j + 1
        f(i) = ( f(i) - f(i - 1) ) / (x(i) - x(i - j) );
    end
end
end
```

CODICE 4.7

```
% [ p ] = Ex_4_7(x, f, xx)
% Algoritmo generalizzato di Horner: dato un set di punti, valuta il polinomio
% su essi.
% Input:
% - x: vettore contenente le ascisse di interpolazione
% - f: vettore contenente le differenze divise
% - xx: vettore di punti sui quali valutare il polinomio
% Output:
% - p: vettore contenente le valutazioni del polinomio sui punti contenuti
%      in xx.
function [ p ] = Ex_4_7(x, f, xx)
n = length(x) - 1;
j = length(xx);
p = zeros(j, 1);

for i = 1 : j
    p(i) = f(n + 1);
    for k = n : -1 : 1
        p(i) = p(i) * (xx(i) - x(k)) + f(k);
    end
end
end
```

CODICE 4.8

```
% [ f ] = Ex_4_8(x, f)
% Algoritmo per il calcolo della differenza divisa di Hermite.
% Input:
%   - x: vettore contenente le 2n+2 ascisse (n = grado polinomio)
%   - f: vettore contenente f(x0), f'(x0), f(x1), f'(x1)...f(xn), f'(xn)
% Output:
%   - f: vettore contenente i valori delle differenze divise calcolate
function[ f ] = Ex_4_8(x, f)
n = length(x);
for i = (n - 1) : -2 : 3
    f(i) = ( f(i) - f(i - 2) ) / ( x(i) - x(i - 1) );
end
for j = 2 : n - 1
    for i = n : -1 : j + 1
        f(i) = ( f(i) - f(i - 1) ) / ( x(i) - x(i - j) );
    end
end
end
end
```

CODICE 4.9

```
f = @(x) (x - 1).^9; % la funzione
xi = linspace(0, 1, 5); % le ascisse interpolanti la funzione
fi = f(xi); % i corrispondenti valori delle ascisse interpolanti

ddN = Ex_4_6(xi, fi); % differenze divise di Newton

xiH = reshape([xi ; xi],1,[]); % le 2n+2 ascisse di Hermite
syms x;
fSyms = (x - 1).^9;
df = matlabFunction(diff(fSyms)); % derivata della funzione
fiH = zeros(1, length(xiH)); % vettore che conterra' f(x0), f'(x0), f(x1),
f'(x1)...f(xn), f'(xn) per Hermite
for i = 1 : length(xiH)
    if mod(i, 2) == 1
        fiH(i) = f(xiH(i));
    else
        fiH(i) = df(xiH(i));
    end
end
ddH = Ex_4_8(xiH, fiH); % differenze divise di Hermite

x = linspace(0, 1, 101); % linspace richiesto

pN = Ex_4_7(xi, ddN, x); % valutazioni tramite Horner generalizzato del
polinomio di Newton sul linspace
pH = Ex_4_7(xiH, ddH, x); % valutazioni tramite Horner generalizzato del
polinomio di Hermite sul linspace

% parte grafica
% Newton
figure;
hold on;
grid on;
fplot(f, [0, 1], 'b');
plot(x, pN, 'r');
plot(xi, fi, 'r*');
```



```

legend('funzione', 'polinomio', 'ascisse interpolanti', 'location','southeast');
title('Polinomio interpolante di Newton');

% Hermite
figure;
hold on;
grid on;
fplot(f, [0, 1], 'b');
set(findobj(gca, 'Type', 'Line', 'Color', 'b'), 'LineWidth', 3);
plot(x, pH, 'g', 'LineWidth', 1.5);
plot(xi, fi, 'g*', 'Markersize', 8);
legend('funzione', 'polinomio', 'ascisse interpolanti', 'location','southeast');
title('Polinomio interpolante di Hermite');
plot(x, pN, 'Linestyle', 'none');

% Confronto tra i due polinomi
figure;
hold on;
grid on;
fplot(f,[0, 1],'b');
plot(x, pN, 'r.', 'Markersize', 7);
plot(x, pH, 'g.', 'Markersize', 7);
plot(xi, fi, 'kO');
legend('funzione', 'Newton', 'Hermite', 'ascisse interpolanti',
'location','southeast');
title('Polinomi di Newton ed Hermite a confronto');

```

CODICE 4.11

```

% [ py, e ] = Ex_4_11(f, a, b, n, xType, g)
%   Algoritmo per valutare e graficare l'approssimazione polinomiale di una
%   funzione rispetto alla funzione stessa.
% Input:
%   - f: la funzione
%   - a: estremo sinistro dell'intervallo desiderato
%   - b: estremo destro dell'intervallo desiderato
%   - n: numero di intervalli desiderati tra a e b che definiscono la
%   partizione delle ascisse interpolanti
%   - xType: true per calcolare e valutare il polinomio usando le ascisse
%   di Chebyshev, false per usare una partizione uniforme
%   - g: true per abilitare la parte grafica, false altrimenti
% Output:
%   - py: valori assunti dal polinomio interpolante la funzione sul
%   linspace di 10001 punti generato dal metodo
%   - e: norma infinito di ||f - p||
function[ py, e ] = Ex_4_11(f, a, b, n, xType, g)
if xType % ascisse di Chebyshev
    x = zeros(1, n + 1);
    for i = n + 1 : -1 : 1
        x(1, i) = (a + b)/2 + (b - a)/2 * cos(((2 * (n - i + 1) + 1)* pi)/(2 *
(n + 1)));
    end
    y = f(x); % valori della funzione nelle ascisse di Chebyshev
else
    x = linspace(a, b, n + 1); % partizione uniforme di n + 1 ascisse
    y = f(x); % valori della funzione in corrispondenza della partizione
end
xx = linspace(a, b, 10001);
py = Ex_4_7(x, Ex_4_6(x, y), xx)'; % valori assunti dal polinomio interpolante
sul linspace
if g

```

```

    hold on;
    grid on;
    title(sprintf('n = %d', n));
    fplot(f, [a, b], 'b');
    plot(xx, py, 'r');
    plot(x, y, 'r.', 'Markersize', 8);
end
e = norm(py - f(xx), inf);
end

```

CODICE 4.11 SCRIPT

```

clc;
Runge = @(x) 1./(1+x.^2);
Bernstein = @(x) abs(x);
Pdf = @(x) 1./(x.^4 - x.^2 + 1);
py = zeros(8, 10001);
e = zeros(8, 3);
grafica = true; % true per abilitare la parte grafica
Chebyshev = false; % true per usare le ascisse di Chebyshev
for i = 1 : 3
    if i == 1
        f = Runge;
        a = -5;
        b = 5;
        string = 'di Runge';
    else if i == 2
        f = Bernstein;
        a = -1;
        b = 1;
        string = 'di Bernstein';
    else
        f = Pdf;
        a = -5;
        b = 5;
        string = 'del Pdf';
    end
end
if grafica
    figure('name', sprintf('Esempio %s',string), 'numbertitle', 'off');
end
for j = 1 : 8
    if grafica
        subplot(2, 4, j);
    end
    [p , err ] = Ex_4_11(f, a, b, 5*j, Chebyshev, grafica);
    py(j, :) = p;
    e(j, i) = err;
end
if grafica
    figure('name', sprintf('Comparazione errori %s',string), 'numbertitle',
'off');
    hold on;
    grid on;
    xx = linspace(a, b, 10001);
    plot(xx , abs( f(xx) - py(1,:) ) , 'b');
    plot(xx , abs( f(xx) - py(2,:) ) , 'r');
    plot(xx , abs( f(xx) - py(3,:) ) , 'g');
    legend('n = 5', 'n = 10', 'n = 15', 'location', 'north');
    title(sprintf('Comparazione degli errori - Esempio %s',string));
end

```

```

end
disp('          Runge          Bernstein          Pdf');
for i = 1 : 8
    fprintf('n = %d',i*5);
    for j = 1 : 3
        fprintf('\t\t%8.4f\t\t',e(i,j));
    end
    fprintf('\n');
end
end

```

CODICE 4.16.1

```

% [ m ] = Ex_4_16_1(phi, xi, dd)
%   Algoritmo per il calcolo del vettore m del sistema lineare tridiagonale
%   per determinare i fattori m(i) nell'espressione della spline cubica
%   naturale.
% Input:
%   - phi: vettore dei fattori phi sulla matrice dei coefficienti (len = n - 1)
%   - xi: vettore dei fattori xi sulla matrice dei coefficienti (len = n - 1)
%   - dd: vettore delle differenze divise (len = n - 1)
% Output:
%   - m: vettore contenente le m(i) calcolate (len = n + 1)
function [ m ] = Ex_4_16_1(phi, xi, dd)
n = length(xi) + 1;
u = zeros(1, n - 1);
l = zeros(1, n - 2);
u(1) = 2;
for i = 2 : n - 1
    l(i) = phi(i) / u(i - 1);
    u(i) = 2 - l(i) * xi(i - 1);
end
dd = 6 * dd;
y = zeros(1, n - 1);
y(1) = dd(1);
for i = 2 : n - 1
    y(i) = dd(i) - l(i) * y(i - 1);
end
m = zeros(1, n - 1);
m(n - 1) = y(n - 1) / u(n - 1);
for i = n - 2 : - 1 : 1
    m(i) = (y(i) - xi(i) * m(i + 1)) / u(i);
end
m = [0 m 0]; % condizioni della spline naturale: m(0) = m(n) = 0
end

```

CODICE 4.16.2

```

% [ f ] = Ex_4_16_2(x, f)
%   Calcolo del vettore delle differenze divise per la risoluzione del
%   sistema tridiagonale lineare delle spline cubiche.
% Input:
%   - x: vettore delle ascisse d'interpolazione
%   - f: vettore dei valori assunti dalla funzione in tali ascisse
% Output:
%   - f: vettore contenente i valori delle differenze divise per il calcolo
%       del sistema lineare tridiagonale per le spline cubiche
function [ f ] = Ex_4_16_2(x, f)
n = length(x) - 1;
for j = 1 : 2
    for i = n + 1 : - 1 : j + 1

```

```

        f(i) = ( f(i) - f(i - 1) ) / (x(i) - x(i - j) );
    end
end
f = f(3 : length(f));
end

```

CODICE 4.17

```

% [ m ] = Ex_4_17(phi, xi, dd)
%   Algoritmo per il calcolo del vettore m del sistema lineare tridiagonale
%   per determinare i fattori m(i) nell'espressione della spline cubica
%   not a knot.
% Input:
%   - phi: vettore dei fattori phi sulla matrice dei coefficienti (len = n - 1)
%   - xi: vettore dei fattori xi sulla matrice dei coefficienti (len = n - 1)
%   - dd: vettore delle differenze divise (len = n - 1)
% Output:
%   - m: vettore contenente le m(i) calcolate (len = n + 1)
function [ m ] = Ex_4_17( phi, xi, dd )
n = length(xi) + 1;
if n + 1 < 4
    error('ATTENZIONE: il numero di ascisse interpolanti è inferiore a 4, questo
significa che la spline not a knot interpolante la funzione coincide con la
funzione stessa');
end
dd = [6 * dd(1); 6 * dd; 6 * dd(length(dd))];
w = zeros(n, 1);
u = zeros(n + 1, 1);
l = zeros(n, 1);

y = zeros(n + 1, 1);
m = zeros(n + 1, 1);

u(1) = 1;
w(1) = 0;

l(1) = phi(1);
u(2) = 2 - phi(1);
w(2) = xi(1) - phi(1);

l(2) = phi(2) / u(2);
u(3) = 2 - ( l(2) * w(2) );
w(3) = xi(2);

for i = 4 : n - 1
    l(i - 1) = phi(i - 1) / u(i - 1);
    u(i) = 2 - l(i - 1) * w(i - 1);
    w(i) = xi(i - 1);
end

l(n - 1) = ( phi(n - 1) - xi(n - 1) ) / u(n - 1);
u(n) = 2 - xi(n - 1) - l(n - 1) * w(n - 1);
w(n) = xi(n - 1);

l(n) = 0;
u(n + 1) = 1;

y(1) = dd(1);
for i = 2 : n + 1
    y(i) = dd(i) - l(i - 1) * y(i - 1);

```

```

end

m(n + 1) = y(n + 1) / u(n + 1);
for i = n : -1 : 1
    m(i) = (y(i) - w(i) * m(i + 1))/u(i);
end

m(1) = m(1) - m(2) - m(3);
m(n + 1) = m(n + 1) - m(n) - m(n - 1);

end

```

CODICE 4.18

```

% [ s ] = Ex_4_18(x, f, m)
%   Algoritmo per la determinazione dell'espressione, polinomiale a tratti,
%   della spline cubica.
% Input:
%   - x: vettore delle ascisse d'interpolazione
%   - f: vettore dei valori assunti nelle ascisse d'interpolazione
%   - m: vettore delle m
% Output:
%   - s: vettore dei polinomi costituenti la spline polinomiale a tratti
function [ s ] = Ex_4_18( xi, fi, m)
n = length(xi) - 1;
s = sym('x' , [n 1]);
syms x;
for i = 2 : n + 1
    hi = xi(i) - xi(i - 1);
    ri = fi(i - 1) - hi^2/6 * m(i - 1);
    qi = (fi(i) - fi(i - 1))/hi - hi/6 * (m(i) - m(i - 1));
    s(i - 1) = ( (x - xi(i - 1))^3 * m(i) + (xi(i) - x)^3 * m(i - 1) ) / (6 *
hi) + qi * (x - xi(i - 1)) + ri;
end
end

```

CODICE 4.19

```

% [ s ] = Ex_4_19(x, f, type)
%   Algoritmo per la determinazione degli n polinomi che formano una spline
%   cubica di tipo naturale (type = false) o not a knot (type = true).
% Input:
%   - x: vettore delle ascisse d'interpolazione
%   - f: vettore dei valori assunti nelle ascisse d'interpolazione
%   - type: true se la spline e' di tipo not a knot, false se
%           e' di tipo naturale
% Output:
%   - s: il vettore contenente le n espressioni dei polinomi costituenti
%       la spline
function [ s ] = Ex_4_19(x, f, type)
n = length(x) - 1;
xi = zeros(1, n - 1);
phi = zeros(1, n - 1);
for i = 1 : n - 1
    phi(i) = ( x(i + 1) - x(i) ) / ( x(i + 2) - x(i) );
    xi(i) = ( x(i + 2) - x(i + 1) ) / ( x(i + 2) - x(i) );
end
dd = Ex_4_16_2(x, f);
if type

```

```

        m = Ex_4_17(phi, xi, dd);
    else
        m = Ex_4_16_1(phi, xi, dd);
    end
    s = Ex_4_18(x, f, m);
end

```

CODICE 4.20 SCRIPT

```

clc;
Runge = @(x) 1./(1+x.^2);
Bernstein = @(x) abs(x);
Pdf = @(x) 1./(x.^4 - x.^2 + 1);
for k = 1 : 2
    for i = 1 : 3
        if i == 1
            f = Runge;
            a = -5;
            b = 5;
            if k == 1
                figure('name', 'Esempio di Runge: spline not-a-knot');
            else
                figure('name', 'Esempio di Runge: spline naturale');
            end
        else if i == 2
            f = Bernstein;
            a = -1;
            b = 1;
            if k == 1
                figure('name', 'Esempio di Bernstein: spline not-a-knot');
            else
                figure('name', 'Esempio di Bernstein: spline naturale');
            end
        else
            f = Pdf;
            a = -5;
            b = 5;
            if k == 1
                figure('name', 'Esempio del Pdf: spline not-a-knot');
            else
                figure('name', 'Esempio del Pdf: spline naturale');
            end
        end
    end
    for j = 1 : 4
        subplot(2, 2, j);
        Ex_4_20_Grafica(f, a, b, 10 * j, k);
    end
end
end
Ex_4_20_ErrTab;
clear stop; clear string; clear i; clear j; clear k; clear a; clear b; clear
nak; clear f;

```

CODICE 4.20 GRAFICA

```
% [ xx ] = Ex_4_20_Grafica(f, a, b, n, nak)
%   Algoritmo per valutare e graficare una spline rispetto ad una funzione.
% Input:
%   - f: la funzione
%   - a: estremo sinistro dell'intervallo desiderato
%   - b: estremo destro dell'intervallo desiderato
%   - n: numero di intervalli desiderati tra a e b che definiscono la
%   partizione delle ascisse interpolanti
%   - nak: true per graficare una spline di tipo not a knot, false altrimenti
function[] = Ex_4_20_Grafica(f, a, b, n, nak)
x = linspace(a, b, n + 1);
xx = linspace(a, b, 1001);
y = f(x);
yy = Ex_4_20(x, Ex_4_19(x, y, nak), xx, true);
hold on;
grid on;
title(sprintf('n = %d', n));
fplot(f, [a, b], 'b');
plot(xx, yy, 'r');
plot(x, y, 'r.', 'MarkerSize', 8);
if nak
    plot(x(2), y(2), 'rO', 'MarkerSize', 7);
    plot(x(length(x) - 1), y(length(y) - 1), 'rO', 'MarkerSize', 7);
end
end
```

CODICE 4.20 ErrTab

```
Runge = @(x) 1./(1+x.^2);
Bernstein = @(x) abs(x);
Complemento = @(x) 1./(x.^4 - x.^2 + 1);
fig = figure('name', 'Errori e tempo di calcolo di ciascuna funzione per ogni
n', 'units', 'normalized', 'numbertitle', 'off', 'Position', [0 0 1 0.27]);
col = {'<html><h1>Runge</h1></html>' , '<html><h1>Bernstein</h1></html>' ,
'<html><h1>Pdf</h1></html>'};
row = {'<html><h1>n = 10<html><h1>' , '<html><h1>n = 20<html><h1>' ,
'<html><h1>n = 30<html><h1>' , '<html><h1>n = 40</h1></html>'};
data = cellstr(char(4,3));
for i = 1 : 3
    if i == 1
        f = Runge;
        string = 'Runge';
        a = -5;
        b = 5;
    else if i == 2
        f = Bernstein;
        string = 'Bernstein';
        a = -1;
        b = 1;
    else
        f = Complemento;
        string = 'Pdf';
        a = -5;
        b = 5;
    end
end
for j = 1 : 4
    tic;
```

```

        x = linspace(a, b, j*10 + 1);
        xx = linspace(a, b, 1001);
        y = f(x);
        matlabSpline = spline(x, y, xx);
        mySpline = Ex_4_20(x, Ex_4_19(x, y, true), xx, false);
        err = norm(mySpline - matlabSpline, inf);
        stop = toc;
        fprintf('Funzione: %s ; n = %d ; Errore = %1.3d ; %1.2f s\n', string,
j*10, err, stop);
        data(j, i) = cellstr(sprintf('      ||e|| = %s , %s s', num2str(err,
'%1.3d'), num2str(stop, '%1.2f')));
    end
end
t = uitable(fig, 'Data', data, 'unit', 'normalized', 'position', [0 0 1 1],
'ColumnName', col, 'RowName', row, 'ColumnWidth', {350}, 'FontSize', 20);

```

CODICE 4.20

```

% [ xx ] = Ex_4_20(p, s, xx, check)
%   Algoritmo che valuta una spline su una data partizione uniforme.
% Input:
%   - p: partizione, ovvero il vettore delle ascisse interpolanti
%   - s: il vettore contenente gli n polinomi che costituiscono la spline
%   - xx: linspace sul quale valutare i polinomi costituenti la spline
%   - check: true per abilitare i controlli sulla correttezza dei parametri
%           in input, false altrimenti
% Output:
%   - xx: vettore di valutazioni della spline in ciascun punto del linspace
function [ xx ] = Ex_4_20(p, s, xx, check)
n = length(p) - 1;
if check
    if (n + 1 ~= length(unique(p)))
        fprintf('\nATTENZIONE: le ascisse interpolanti non sono tutte
distinte!\nVerranno eliminati i doppioni.\n');
        p = unique(p);
        n = length(p) - 1;
    end
    if n < 2
        error('Inserire almeno 3 ascisse interpolanti!');
    end
    p = sort(p);
    a = p(1);
    b = p(n + 1);
    if ~isequal(p, linspace(a, b, n + 1))
        fprintf('\nATTENZIONE: a meno di possibili errori di round off, la
partizione scelta non sembra essere uniforme!\n');
    end
end
j = 1;
k = 1;
for i = 1 : n
    isIn = true;
    while j <= length(xx) && isIn
        if xx(j) >= p(i) && xx(j) <= p(i + 1)
            j = j + 1;
        else
            isIn = false;
        end
    end
    xx(k : j - 1) = subs(s(i), xx(k : j - 1));
    k = j;
end

```



```
end  
end
```

CODICE 4.22

```
clc;  
x = [5.22; 4.00; 4.28; 3.89; 3.53; 3.12; 2.73; 2.70; 2.20; 2.08; 1.94];  
x1 = log(x);  
V = [1 0; 1 1; 1 2; 1 3; 1 4; 1 5; 1 6; 1 7; 1 8; 1 9; 1 10];  
[res, r] = Ex_3_8_1(V, x1);  
alpha = exp(res(1));  
lambda = -res(2);  
fprintf('\nResiduo: %2.5f\nAlpha: %2.5f\nLambda: %2.5f\n', r, alpha, lambda);  
f = @(t) alpha * exp(-lambda * t);  
xi = linspace(0,10,10001);  
hold on;  
fplot(f, [0, 10], 'b');  
plot(0 : 10, x(1 : 11), 'r*');  
legend('Modello teorico', 'Misure sperimentali');
```

CODICE 4.1 COMPLEMENTARE

```
clc;  
st = [6.571; 13.283; 23.137; 36.134; 52.269; 71.551; 93.971; 119.530; 148.236;  
180.072];  
V = [1 1 1; 1 2 4; 1 3 9; 1 4 16; 1 5 25; 1 6 36; 1 7 49; 1 8 64; 1 9 81; 1 10  
100];  
[res] = Ex_3_8_1(V, st);  
fprintf('\nPosizione iniziale: %2.5f\nVelocita' iniziale: %2.5f  
m/s\nAccelerazione: %2.5f m/s^2\n', res(1), res(2), res(3));  
plot(1:10, st, 'b');  
hold on;  
plot(1:10, st, 'r*');  
legend('Modello teorico', 'Misure sperimentali', 'location', 'northwest');
```

CODICE 4.2 COMPLEMENTARE

```
clc;  
x = [0.905; 0.819; 0.741; 0.670; 0.607; 0.549; 0.497; 0.449; 0.407; 0.368];  
x1 = log(x);  
V = [1 1; 1 2; 1 3; 1 4; 1 5; 1 6; 1 7; 1 8; 1 9; 1 10];  
[res] = Ex_3_8_1(V, x1);  
alpha = exp(res(1));  
lambda = -res(2);  
fprintf('\nAlpha: %2.5f\nLambda: %2.5f\n', alpha, lambda);  
f = @(t) alpha * exp(-lambda * t);  
fplot(f, [1, 10], 'b');  
hold on;  
plot(1:10, x(1:10), 'r*');  
legend('Modello teorico', 'Misure sperimentali');
```

CODICI CAPITOLO 5

CODICE 5.4

```
% [ If ] = Ex_5_4( f, a, b, n, g )
%   Algoritmo per il calcolo dell'integrale di una funzione in un dato
%   intervallo [a, b] utilizzando la formula dei trapezi composta.
% Input:
%   - f: funzione
%   - a: estremo sinistro dell'intervallo
%   - b: estremo destro dell'intervallo
%   - n: numero desiderato di partizioni nell'intervallo [a, b]
%   - g: abilita la parte grafica se true, false altrimenti
% Output:
%   -If: l'area approssimata
function [ If ] = Ex_5_4( f, a, b, n, g )
xi = linspace(a, b, n + 1);
k = (b - a) / (2 * n);
f0 = feval(f, a);
fn = feval(f, b);
If = 0;
for i = 1 : n - 1
    If = If + feval(f, xi(i + 1));
end
If = k * (f0 + 2*If + fn);
if g
    figure;
    hold on;
    line(xlim, [0 0], 'Color', 'black');
    area(xi, f(xi));
    leg(1) = plot([a,b],[0 0], 'g0');
    leg(2 : n) = plot(xi(2 : n), 0, 'r*');
    fplot(f,[a - 0.5, b + 0.5]);
    legend(leg(1 : 2),'estremi intervallo','ascisse interpolanti');
    title('Calcolo dell'integrale - Metodo dei trapezi composti');
    shg;
end
end
```

CODICE 5.5

```
% [ If ] = Ex_5_5( f, a, b, n, g )
%   Algoritmo per il calcolo dell'integrale di una funzione in un dato
%   intervallo [a, b] utilizzando la formula di Simpson composta.
% Input:
%   - f: funzione
%   - a: estremo sinistro dell'intervallo
%   - b: estremo destro dell'intervallo
%   - n: numero desiderato di partizioni nell'intervallo [a, b]
%   - g: abilita la parte grafica se true, false altrimenti
% Output:
%   -If: l'area approssimata
function [ If ] = Ex_5_5( f, a, b, n, g )
if mod(n, 2) == 1 % controllo sulla parita' degli intervalli
    n = n + 1;
end
xi = linspace(a, b, n + 1);
k = (b - a) / (3 * n);
```

```

f0 = feval(f, a);
fn = feval(f, b);
res = 0;
for i = 1 : n/2
    res = res + feval(f, xi(2*i));
end
If = 0;
for i = 0 : n/2
    If = If + feval(f, xi(2*i + 1));
end
If = k*(4*res + 2*If - f0 - fn);
if g
    figure;
    hold on;
    line(xlim, [0 0], 'Color', 'black');
    area(xi, f(xi));
    leg(1) = plot([a,b],[0 0], 'g0');
    leg(2 : n) = plot(xi(2 : n), 0, 'r*');
    fplot(f,[a - 0.5, b + 0.5]);
    legend(leg(1 : 2),'estremi intervallo','ascisse interpolanti');
    title('Calcolo dell''integrale - Metodo di Simpson composita');
    shg;
end
end

```

CODICE 5.6

```

% [If, ptx, xi] = Ex_5_6( f, a, b, tol, g)
%   Algoritmo per il calcolo dell'integrale di una funzione in un dato
%   intervallo [a, b] utilizzando la formula dei trapezi adattativa.
% Input:
%   - f: funzione
%   - a: estremo sinistro dell'intervallo
%   - b: estremo destro dell'intervallo
%   - tol: tolleranza desiderata
%   - g: abilita la parte grafica se true, false altrimenti
% Output:
%   - If: l'area approssimata
%   - ptx: numero di punti generati dal metodo per soddisfare la tolleranza
%   data
%   - xi: vettore contenente i punti generati
function [If, ptx, xi] = Ex_5_6( f, a, b, tol, g)
xi = [a b];
fa = feval(f, a);
fb = feval(f, b);
[If, ptx, xi] = AdaptativeTrap( f, a, b, tol, fa, fb, 3, xi);
if g
    figure;
    semilogx(xi, f(xi), '+r','markers',5);
end
end

function [If, ptx, xi] = AdaptativeTrap( f, a, b, tol, fa, fb, ptx, xi)
h = (b - a)/2;
I1 = h*(fa + fb);
m = (a + b)/2;
xi = [xi m];
fm = feval(f, m);
I2 = I1/2 + h*fm;
err = abs(I2 - I1)/3;
if err <= tol

```

```

        If = I2;
else
    [Isx, ptSx, xiSx] = AdaptativeTrap(f, a, m, tol/2, fa, fm, 1, xi);
    [Idx, ptDx, xiDx] = AdaptativeTrap(f, m, b, tol/2, fm, fb, 1, xiSx);
    If = Isx + Idx;
    ptx = ptx + ptSx + ptDx;
    xi = xiDx;
end
return;
end

```

CODICE 5.7

```

% [If, ptx, xi] = Ex_5_7 ( f, a, b, tol, g)
%   Algoritmo per il calcolo dell'integrale di una funzione in un dato
%   intervallo [a, b] utilizzando la formula di Simpson adattativa.
% Input:
%   - f: funzione
%   - a: estremo sinistro dell'intervallo
%   - b: estremo destro dell'intervallo
%   - tol: tolleranza desiderata
%   - g: abilita la parte grafica se true, false altrimenti
% Output:
%   - If: l'area approssimata
%   - ptx: numero di punti generati dal metodo per soddisfare la tolleranza
%   data
%   - xi: vettore contenente i punti generati
function [If, ptx, xi] = Ex_5_7 ( f, a, b, tol, g)
xi = [a, b];
m = (a + b)/2;
fa = feval(f, a);
fm = feval(f, m);
fb = feval(f, b);
[If, ptx, xi] = AdaptativeSimpson( f, a, m, b, tol, fa, fm, fb, 5, xi );
if g
    figure;
    semilogx(xi, f(xi), '+r','markers',5);
end
end

function [If, ptx, xi] = AdaptativeSimpson( f, a, m, b, tol, fa, fm, fb, ptx, xi
)
h = (b - a)/6;
x1 = (a + m)/2;
x3 = (m + b)/2;
xi = [xi, x1, x3];
fx1 = feval(f, x1);
fx3 = feval(f, x3);
I2 = h*(fa + 4*fm + fb);
I4 = 1/2*(I2 + 2*h*(2*fx1 + 2*fx3 - fm));
err = abs(I4 - I2)/15;
if err <= tol
    If = I4;
else
    [IfSx, ptxSx, xiSx] = AdaptativeSimpson( f, a, x1, m, tol/2, fa, fx1, fm, 2,
xi );
    [IfDx, ptxDx, xiDx] = AdaptativeSimpson( f, m, x3, b, tol/2, fm, fx3, fb, 2,
xiSx );
    If = IfSx + IfDx;
    ptx = ptx + ptxSx + ptxDx;
    xi = xiDx;
end

```

```

end
return;
end

```

CODICE 5.9

```

f = @(x) -2.*x.^(-3).*cos(x.^(-2));
If = sin(10^-4) - sin(4);
a = 1/2;
b = 100;
clc;
disp('Errore formula composta trapezi');
for i = 1000 : 1000 : 10000
    fprintf('\n# intervalli = %d\t\terrore = %2.2d', i, (abs(If -
Ex_5_4(f,a,b,i,false))));
end
fprintf('\n\n');
disp('Errore formula trapezi adattativa');
for i = 1 : 4
    [I,n,xi] = Ex_5_6(f, a, b, 10^-i, false);
    fprintf('\nTolleranza = %1.1d\t\t# ascisse = %d\t\terrore = %2.2d',10^-i, n,
abs(If - I));
end
fprintf('\n');

```

CODICE 5.10

```

f = @(x) -2.*x.^(-3).*cos(x.^(-2));
If = sin(10^-4) - sin(4);
a = 1/2;
b = 100;
clc;
disp('Errore formula composta Simpson');
for i = 1000 : 1000 : 10000
    fprintf('\n# intervalli = %d\terrore = %2.2d',i,(abs(If -
Ex_5_5(f,a,b,i,false))));
end
fprintf('\n\n');
disp('Errore formula di Simpson adattativa');
for i = 1 : 5
    [I,n,xi] = Ex_5_7(f, a, b, 10^-i, false);
    fprintf('\nTolleranza = %1.1d\t\t# ascisse = %d\t\terrore = %2.2d',10^-i, n,
abs(If - I));
end
fprintf('\n');

```

CODICE 5 COMPLEMENTARE SCRIPT

```

f = @(x) exp(-(20.*x).^2 );
If = integral(f, -1, 1);
tol = 10^-6;
[areaTrapAdapt,nValTrapAdapt] = Ex_5_6(f, -1, 1, tol, false);
[areaSimpAdapt,nValSimpAdapt] = Ex_5_7(f, -1, 1, tol, false);
[areaTrapComp,nValTrapComp] = Ex_5_Compl_Trap(f, -1, 1, tol, false);
[areaSimpComp,nValSimpComp] = Ex_5_Compl_Simp(f, -1, 1, tol, false);

errTrapAdapt = abs(If - areaTrapAdapt); %stima errore
errSimpAdapt = abs(If - areaSimpAdapt);

```

```

errTrapComp = abs(If - areaTrapComp);
errSimpComp = abs(If - areaSimpComp);
clc;
fprintf('\nArea trapezi adattativa = %2.16f\nValutazioni di funzioni = %d\nStima
dell''errore = %1.2d\n\nArea Simpson adattativa = %2.16f\nValutazioni di
funzioni = %d\nStima dell''errore = %1.2d\n\nArea trapezi composita =
%2.16f\nValutazioni di funzioni = %d\nStima dell''errore = %1.2d\n\nArea Simpson
composita = %2.16f\nValutazioni di funzioni = %d\nStima dell''errore =
%1.2d\n', areaTrapAdapt, nValTrapAdapt, errTrapAdapt, areaSimpAdapt, nValSimpAdapt,
errSimpAdapt, areaTrapComp, nValTrapComp, errTrapComp, areaSimpComp,
nValSimpComp, errSimpComp);

```

CODICE 5 COMPLEMENTARE TRAPEZI

```

% [ I2n, nEval ] = Ex_5_Compl_Trap( f, a, b, tol, g )
%   Algoritmo per il calcolo dell'integrale di una funzione in un dato
%   intervallo [a, b] utilizzando la formula dei trapezi composita a passo
%   costante.
% Input:
%   - f: funzione
%   - a: estremo sinistro dell'intervallo
%   - b: estremo destro dell'intervallo
%   - tol: tolleranza desiderata
%   - g: abilita la parte grafica se true, false altrimenti
% Output:
%   - I2n: l'area approssimata
%   - nEval: numero di valutazioni di funzioni impiegate
function [ I2n, nEval ] = Ex_5_Compl_Trap( f, a, b, tol, g )
n = 1;
err = inf;
ba = b - a;
h = ba/2;
fa = f(a);
fb = f(b);
In = h*(fa + fb);
while err > tol
    n = 2*n;
    x = linspace(a, b, n + 1);
    I2n = In/2 + h * sum(f(x(2 : 2 : n)));
    err = abs(I2n - In)/3;
    In = I2n;
    h = ba/(2*n);
end
nEval = n + 1;
if g
    figure;
    hold on;
    line(xlim, [0 0], 'Color', 'black');
    area(x, f(x));
    leg(1) = plot([a,b],[0 0], 'g0');
    leg(2 : n) = plot(x(2 : n), 0, 'r*');
    fplot(f, [a, b]);
    legend(leg(1 : 2), 'estremi intervallo', 'ascisse interpolanti');
    title('Calcolo dell''integrale - Metodo dei trapezi composti a passo
costante');
    shg;
end
end

```

CODICE 5 COMPLEMENTARE SIMPSON

```
% [ I2n, nEval ] = Ex_5_Compl_Simp( f, a, b, tol, g )
% Algoritmo per il calcolo dell'integrale di una funzione in un dato
% intervallo [a, b] utilizzando la formula di Simpson composta a passo
% costante.
% Input:
% - f: funzione
% - a: estremo sinistro dell'intervallo
% - b: estremo destro dell'intervallo
% - tol: tolleranza desiderata
% - g: abilita la parte grafica se true, false altrimenti
% Output:
% - I2n: l'area approssimata
% - nEval: numero di valutazioni di funzioni impiegate
function [ I2n, nEval ] = Ex_5_Compl_Simp( f, a, b, tol, g )
n = 2;
err = inf;
ba = b - a;
h = ba/6;
m = (a + b)/2;
fa = f(a);
fb = f(b);
sx = f(m);
In = h*(fa + 4*sx + fb);
while err > tol
    n = 2*n;
    x = linspace(a, b, n + 1);
    dx = sum(f(x(2 : 2 : n)));
    I2n = In/2 + h * (sx + dx);
    err = abs(I2n - In)/15;
    In = I2n;
    sx = dx;
    h = ba/(3*n);
end
nEval = n + 3;
if g
    figure;
    hold on;
    line(xlim, [0 0], 'Color', 'black');
    area(x, f(x));
    leg(1) = plot([a,b],[0 0], 'g0');
    leg(2 : n) = plot(x(2 : n), 0, 'r*');
    fplot(f,[a, b]);
    legend(leg(1 : 2),'estremi intervallo','ascisse interpolanti');
    title('Calcolo dell''integrale - Metodo dei trapezi composti a passo
costante');
    shg;
end
end
```

CODICI CAPITOLO 6

CODICE 6.2.1

```
clc;
tol = 10^-5;
format long;

disp('Tolleranza di 10^-5');
for n = 3 : 3 : 30
    [A, COL] = Ex_6_2_3(n);
    lambda = Ex_6_2_2(A, COL, tol);
    approx = 2*(1 + cos(pi/(n + 1)));
    err = abs(lambda - approx);
    fprintf('\nn = %d, lambda = %2.6f, approx = %2.6f\ne = %2.6f\n', n, lambda,
approx, err);
end

tol = 10^-10;
fprintf('\n\nTolleranza di 10^-10');

for n = 3 : 3 : 30
    [A, COL] = Ex_6_2_3(n);
    lambda = Ex_6_2_2(A, COL, tol);
    approx = 2*(1 + cos(pi/(n + 1)));
    err = abs(lambda - approx);
    fprintf('\nn = %d, lambda = %2.11f, approx = %2.11f\ne = %2.11f\n', n
, lambda, approx, err);
end
```

CODICE 6.2.2

```
% [ l1 ] = Ex_6_2_2(A, COL, tol)
% Implementazione del metodo delle potenze per il calcolo dell'autovalore
% dominante semplice di una data matrice. Il metodo e' modificato: riceve
% in input una matrice compressa ed usa il MATVEC per colonne.
% Input:
% - A: matrice compressa
% - COL: matrice degli indici di A
% - tol: tolleranza desiderata
% Output:
% - l1: approssimazione dell'autovalore dominante semplice di A
function [ l1 ] = Ex_6_2_2(A, COL, tol)
n = size(A,1);
z = rand(n,1);
l1 = inf;
err = inf;
while err > tol
    q = z / norm(z);
    z = Ex_6_2_5(A, COL, q);
    l0 = l1;
    l1 = q'*z;
    err = abs(l0-l1);
end
end
```

CODICE 6.2.3

```
% [A, COL] = Ex_6_2_3(n)
% Funzione che crea la matrice A secondo le specifiche della traccia
% dell'esercizio 6.2. Prima di restituirla, la comprime.
% Input:
% - n: dimensione desiderata della matrice
% Output:
% - A: matrice compressa
% - COL: matrice degli indici di A
function [A, COL] = Ex_6_2_3(n)
A = 2*eye(n);
for i = 2 : n
    A(i, i-1) = -1;
    A(i-1, i) = -1;
end
[A, COL] = Ex_6_2_4(A);
end
```

CODICE 6.2.4

```
% [B, COL] = Ex_6_2_4(A)
% Funzione che realizza la compressione per righe della matrice in input.
% La dimensione delle matrici in output è stata preallocata per evitare
% l'overhead dovuto alla riallocazione di memoria.
% Input:
% - A: matrice
% Output:
% - B: matrice compressa
% - COL: matrice degli indici di A
function [B, COL] = Ex_6_2_4(A)
n = size(A, 1);
k = 1;
% Calcolo della dimensione massima che potrà assumere B e COL
count = 0;
max = 0;
for i = 1 : n
    for j = 1 : n
        if A(i, j) ~= 0
            count = count + 1;
        end
    end
    if count > max
        max = count;
    end
    count = 0;
end
B = zeros(n, max); % Preallocazione
COL = ones(n, max);
for i = 1 : n
    for j = 1 : n
        if A(i, j) ~= 0
            B(i, k) = A(i, j);
            COL(i, k) = j;
            k = k + 1;
        end
    end
    k = 1;
end
end
```

CODICE 6.2.5

```
% [x] = Ex_6_2_5(A, COL, b)
%   MATVEC per colonne: esegue il prodotto matrice-vettore.
% Input:
%   - A: matrice compressa
%   - COL: matrice degli indici di A
%   - b: vettore
% Output:
%   - b: prodotto
function [x] = Ex_6_2_5(A, COL, b)
[m, n] = size(A);
x = zeros(m, 1);
for j = 1 : n
    for i = 1 : m
        x(i) = x(i) + A(i,j)*b(COL(i,j));
    end
end
end
```

CODICE 6.12.1

```
clc;
for n = 3 : 3: 30
    sol = ones(n, 1)';
    tol = 10^-1; % fissiamo la tolleranza desiderata
    while tol > 10^-10 % costruiamo la A richiesta
        A = eye(n);
        A(1,n) = -1/2;
        for i = 2 : n
            A(i, i - 1) = -1;
        end
        b = [1/2, zeros(1,n-1)]; % costruiamo il vettore dei termini noti
        k = norm(A, inf) * norm(inv(A), inf); % calcolo fattore condizionamento
        A
        err = tol/k;
        err = err * norm(b, inf); % tolleranza da dare in input
        [A, COL] = Ex_6_2_4(A); % compressione matrice
        x = rand(1, n); % randomizzazione innesco
        [solJ, itJ] = Ex_6_12_2(A, COL, b, x, err, false);
        [solGS, itGS] = Ex_6_12_2(A, COL, b, x, err, true);
        fprintf('\nn = %d, tol = %.1d, # it Jac = %d, ||e|| Jac = %.1d, # it GS
= %d, ||e|| GS = %.d\n', n, tol, itJ, norm(sol - solJ, inf), itGS, norm(sol -
solGS, inf));
        tol = tol / 10;
    end
end
```

CODICE 6.12.2

```
% [x, it] = Ex_6_12_2(A, COL, b, x, tol, type)
%   Questa funzione definisce la procedura iterativa di risoluzione del
%   sistema Ax = b secondo il metodo di Jacobi se type e' false, di
%   Gauss-Seidel se type e' true. Il metodo e' ottimizzato
%   sfruttando algoritmi per la compressione di matrici sparse.
% Input:
%   - A: matrice dei coefficienti compressa
%   - COL: matrice degli indici di A
%   - b: vettore dei termini noti
```

```

% - x: approssimazione iniziale
% - tol: tolleranza desiderata
% - type: true per Gauss-Seidel, false per Jacobi
% Output:
% - x: vettore soluzione del sistema
% - i: numero di iterazioni
function [x, i] = Ex_6_12_2(A, COL, b, x, tol, type)
resP = Ex_6_12_4(A, COL, x) - b;
if type
    solve = @(A, COL, resP) Ex_6_12_5(A, COL, resP);
else
    solve = @(A, COL, resP) Ex_6_12_4(A, COL, resP);
end
i = 0;
while norm(resP) > tol
    resP = solve(A, COL, resP);
    x = x - resP;
    resP = Ex_6_12_3(A, COL, x) - b;
    i = i + 1;
end
end

```

CODICE 6.12.3

```

% [x] = Ex_6_12_3(A, COL, b)
% MATVEC per righe: esegue il prodotto matrice-vettore.
% Input:
% - A: matrice compressa
% - COL: matrice degli indici di A
% - b: vettore
% Output:
% - b: prodotto
function [x] = Ex_6_12_3(A, COL, b)
[m, n] = size(A);
x(1 : m) = 0;
for i = 1 : m
    for j = 1 : n
        x(i) = x(i) + A(i,j)*b(COL(i,j));
    end
end
end
end

```

CODICE 6.12.4

```

% [b] = Ex_6_12_4(A, COL, b)
% Funzione che risolve un sistema diagonale con una matrice dei
% coefficienti compressa.
% Input:
% - A: matrice compressa
% - COL: matrice degli indici di A
% - b: vettore dei termini noti
% Output:
% - b: vettore delle soluzioni del sistema
function [b] = Ex_6_12_4(A, COL, b)
[m, n] = size(A);
for i = 1 : m
    for j = 1 : n
        if COL(i, j) == i
            b(i) = b(i)/A(i, j);
        end
    end
end

```

```

end
end
end

```

CODICE 6.12.5

```

% [b] = Ex_6_12_5(A, COL, b)
% Funzione che risolve un sistema triangolare inferiore a diagonale unitaria.
% L'implementazione e' ottimizzata per la tipologia di matrici proposte
% nella traccia dell'esercizio 6.12, ovvero aventi diagonale principale
% pari a 1. Non viene eseguita, fuori dal ciclo interno, la divisione
% per l'i-esimo elemento sulla diagonale, guadagnando in velocita' a
% scapito della generalita'.
% Input:
% - A: matrice compressa
% - COL: matrice degli indici di A
% - b: vettore dei termini noti
% Output:
% - b: vettore soluzione del sistema
function [b] = Ex_6_12_5(A, COL, b)
[m, n] = size(A);
for i = 1 : m
    for j = 1 : n
        if COL(i, j) < i
            b(i) = b(i) - A(i, j)*b(COL(i, j));
        end
    end
end
end
end

```

CODICE 6.1 COMPLEMENTARE

```

clc;
tol = 10^-5;
for i = 1 : 3
    if i == 1
        n = 10;
    else if i == 2
        n = 50;
    else
        n = 100;
    end
end
A = eye(n);
x = zeros(n, 1);
b = zeros(n, 1);
for j = 1 : n
    b(j) = 1/n;
    for k = 1 : n
        if j~=k
            A(j,k)= -1/n;
        end
    end
end
k = norm(A, inf) * norm(inv(A), inf);
err = tol/k;
err = err * norm(b, inf);
[xJac, iJac, eJac] = Ex_6_2Complementare(A, b, x, err, false);
[xGS, iGS, eGS] = Ex_6_2Complementare(A, b, x, err, true);
figure;

```

```

hold on;
semilogy(eJac, 'r');
semilogy(eGS, 'g');
legend('Jacobi', 'Gauss-Seidel');
title(sprintf('Jacobi e Gauss-Seidel: comparazione errori con n = %d', n));
xlabel('# iterazioni');
ylabel('log10||r||');
end

```

CODICE 6.2 COMPLEMENTARE

```

% [ x, i, rVect ] = Ex_6_2Complementare(A, b, x, tol, type)
% Questa funzione definisce la procedura iterativa di risoluzione del
% sistema Ax = b secondo il metodo di Jacobi se type = false, Gauss-Seidel
% se type = true. Il criterio d'arresto e' stato modificato introducendo
% anche una soglia massima di iterazioni funzionale a quanto richiesto
% dalla traccia dell'esercizio.
% Input:
% - A: matrice dei coefficienti
% - b: vettore dei termini noti
% - x: approssimazione iniziale
% - tol: tolleranza desiderata
% - type: true per Gauss-Seidel, false per Jacobi
% Output:
% - x: vettore soluzione del sistema
% - i: numero di iterazioni
% - rVect: vettore delle errori
function [x, i, rVect] = Ex_6_2Complementare(A, b, x, tol, type)
n = length(x);
itMax = 10 * n - 1;
rVect = zeros(10*n, 1);
resP = A*x - b;
rVect(1) = 1/n;
if type
    solve = @(A, resP) Ex_3_1_2(A, resP);
else
    solve = @(A, resP) Ex_3_3_3(A, resP);
end
i = 0;
while norm(resP) > tol && i < itMax
    resP = solve(A, resP);
    x = x - resP;
    resP = A*x - b;
    i = i + 1;
    rNorm = norm(resP, inf);
    rVect(i+1) = rNorm;
end
end

```