



Elaborato di  
**Calcolo Numerico**  
Anno Accademico 2018/2019

Alessio Falai - 6134275 [alessio.falai@stud.unifi.it](mailto:alessio.falai@stud.unifi.it)  
Leonardo Calbi - 6155786 [leonardo.calbi@stud.unifi.it](mailto:leonardo.calbi@stud.unifi.it)

May 25, 2019

# Capitoli

<b>1</b>	<b>Capitolo 1</b>	<b>1</b>
1.1	Esercizio 1 . . . . .	1
1.2	Esercizio 2 . . . . .	2
1.3	Esercizio 3 . . . . .	3
1.4	Esercizio 4 . . . . .	4
<b>2</b>	<b>Capitolo 2</b>	<b>5</b>
2.1	Esercizio 5 . . . . .	5
2.2	Esercizio 6 . . . . .	8
2.3	Esercizio 7 . . . . .	9
<b>3</b>	<b>Capitolo 3</b>	<b>12</b>
3.1	Esercizio 8 . . . . .	12
3.2	Esercizio 9 . . . . .	12
3.3	Esercizio 11 . . . . .	13
3.4	Esercizio 12 . . . . .	14
<b>4</b>	<b>Capitolo 4</b>	<b>15</b>
4.1	Esercizio 14 . . . . .	15
4.2	Esercizio 15 . . . . .	16
4.3	Esercizio 16 . . . . .	17
4.4	Esercizio 17 . . . . .	18
4.5	Esercizio 18 . . . . .	19
4.6	Esercizio 19 . . . . .	20
4.7	Esercizio 20 . . . . .	26
<b>5</b>	<b>Capitolo 5</b>	<b>29</b>
5.1	Esercizio 22 . . . . .	29
5.2	Esercizio 23 . . . . .	30
<b>6</b>	<b>Capitolo 6</b>	<b>32</b>
6.1	Esercizio 24 . . . . .	32
6.2	Esercizio 25 . . . . .	33
6.3	Esercizio 26 . . . . .	34
6.4	Esercizio 27 . . . . .	35
6.5	Esercizio 28 . . . . .	36

# 1 Capitolo 1

## 1.1 Esercizio 1

Verificare che, per  $h$  sufficientemente piccolo:

$$\frac{3}{2}f(x) - 2f(x-h) + \frac{1}{2}f(x-2h) = hf'(x) + O(h^3)$$

---

Usando gli sviluppi di Taylor fino al secondo ordine otteniamo:

$$f(x) = f(x_0) + f'(x_0)h + \frac{1}{2}f''(x_0)h^2 + O(h^3)$$

$$f(x-h) = f(x_0) + f'(x_0)(x-h-x_0) + \frac{1}{2}f''(x_0)(x-h-x_0)^2 + O(h^3)$$

$$f(x-2h) = f(x_0) + f'(x_0)(x-2h-x_0) + \frac{1}{2}f''(x_0)(x-2h-x_0)^2 + O(h^3)$$

Dato che  $x - x_0 = h$ , ponendo  $x_0 = x - h$ , si ha che:

$$f(x-h) = f(x_0) + O(h^3)$$

$$f(x-2h) = f(x_0) - f'(x_0)h + \frac{1}{2}f''(x_0)h^2 + O(h^3)$$

Effettuando le opportune sostituzioni, la relazione iniziale diventa:

$$\frac{3}{2}f(x_0) + \frac{3}{2}f'(x_0)h + \frac{3}{4}f''(x_0)h^2 - 2f(x_0) + \frac{1}{2}f(x_0) - \frac{1}{2}f'(x_0)h + \frac{1}{4}f''(x_0)h^2 + O(h^3) = hf'(x) + O(h^3)$$

Semplificando, si ottiene che:

$$f'(x_0)h + f''(x_0)h^2 + O(h^3) = hf'(x) + O(h^3)$$

A questo punto, scriviamo lo sviluppo di Taylor anche per  $f'(x)$ :

$$f'(x) = f'(x_0) + f''(x_0)h + O(h^2)$$

Infine, otteniamo che:

$$f'(x_0)h + f''(x_0)h^2 + O(h^3) = f'(x_0)h + f''(x_0)h^2 + O(h^3)$$

Dunque, l'uguaglianza iniziale è verificata.

## 1.2 Esercizio 2

Quanti sono i numeri di macchina normalizzati della doppia precisione IEEE? Argomentare la risposta.

---

Nella doppia precisione IEEE si utilizzano 64 bit per rappresentare un numero in virgola mobile in macchina. Di questi, il primo bit è riservato al segno  $\pm$ , i 52 bit successivi rappresentano la mantissa  $\rho$  e i restanti 11 l'esponente  $\eta$ . Un numero reale può essere rappresentato in macchina mediante la formula  $r = \pm \rho \eta$ , con  $\rho = \sum_{i=1}^m \alpha_i b^{1-i}$  e  $\eta = b^{e-\nu}$ . Nel caso dei numeri normalizzati, abbiamo delle restrizioni sui numeri rappresentabili in macchina:

- La mantissa è assunta della forma  $1.f$
- Lo shift  $\nu$  è pari a 1023
- Il valore  $e$  deve essere compreso tra 0 e 2047, estremi esclusi

Dunque, per ricavare il numero di numeri normalizzati della doppia precisione IEEE, contiamo le possibili combinazioni di bit, date le condizioni sopra definite:

- Per il segno abbiamo solo due possibilità,  $+$  oppure  $-$
- Per la mantissa abbiamo esattamente  $2^{52}$  opzioni
- Per il valore di  $e$  abbiamo esattamente 2046 opzioni

Dato che lo shift  $\nu$  non cambia il numero di combinazioni possibili, moltiplicando i dati sopra ottenuti si ottiene:

$$2 \times 2^{52} \times 2046 = 2^{53} \times 2046 = 18\,428\,729\,675\,200\,069\,632$$

### 1.3 Esercizio 3

Eseguire il seguente *script* Matlab:

```
1      format long e
2      n=75;
3      u=1e-300;for i=1:n,u=u*2;end,for i=1:n,u=u/2;end,u
4      u=1e-300;for i=1:n,u=u/2;end,for i=1:n,u=u*2;end,u
```

Spiegare i risultati ottenuti.

---

I risultati ottenuti:

```
1  u =
2      1.0000000000000000e-300
3
4  u =
5      1.119916342203863e-300
```

L'obiettivo del programma proposto è verificare, dopo una serie di operazioni, che il numero restituito sia uguale a quello di partenza e che quindi non si abbia perdita di informazione.

Prima di tutto viene impostato il formato di output a *longE*, formato long decimal (15 cifre dopo la virgola) con notazione scientifica.

Alcune premesse:

In caso di underflow, ovvero quando un calcolo produce un valore più piccolo di *realmin*, in accordo con l'opzione dello standard IEEE, in MATLAB i numeri vengono denormalizzati, perdono quindi il primo 1 sottinteso e sono definiti dalla loro rappresentazione binaria. Inoltre, il più piccolo numero positivo denormalizzato è  $0,494 \times 10^{-323}$ . Ogni risultato più piccolo di questo è posto a zero.

Alcune notazioni importanti:

- *realmin* è il minimo numero normalizzato rappresentabile in MATLAB
- *realmax* è il massimo numero normalizzato rappresentabile in MATLAB
- *eps* è la precisione di macchina

I primi 2 for effettuano prima 75 moltiplicazioni per 2 del numero *u* e poi lo stesso numero di divisioni per 2 di *u*, mantenendone il valore al di sopra di *realmin* e facendo sì che non ci sia perdita di informazione (underflow). I for successivi invece, a causa dello svolgersi prima delle 75 divisioni e poi delle 75 moltiplicazioni, portano alla denormalizzazione del numero e quindi a una sua approssimazione, che risulta nel valore finale di *u*.

## 1.4 Esercizio 4

Eeguire le seguenti istruzioni Matlab:

```
1 format long e
2 a=1.1111111111111111
3 b=1.1111111111111111
4 a+b
5 a-b
```

Spiegare i risultati ottenuti.

I risultati ottenuti:

```
1 a =
2     1.1111111111111111e+00
3
4 b =
5     1.1111111111111110e+00
6
7 ans =
8     2.22222222222221e+00
9
10 ans =
11     8.881784197001252e-16
```

L'obiettivo del programma proposto è verificare se i risultati dati da operazioni di somma e sottrazione siano soggetti a cancellazione numerica o meno.

Innanzitutto, come nel precedente esercizio, viene impostato il formato di output a *longE*, formato long decimal (15 cifre dopo la virgola) con notazione scientifica.

Dopodichè, osservando gli output ottenuti, l'operazione di somma risulta ben condizionata, mentre l'operazione di sottrazione di due numeri molto vicini tra loro risulta invece mal condizionata. Tale malcondizionamento deriva dal fatto che in macchina la semplice operazione  $a - b$  viene effettuata come  $fl(fl(a) - fl(b))$ , dove con  $fl(x)$  indichiamo il valore floating point del numero  $x$ .

Dunque, il numero  $a - b = realmin + eps \times 4$ , mentre invece dovrebbe essere  $10^{-15}$ . Calcolando i vari valori floating, otteniamo:

$$fl(a) = 1,111\,111\,111\,111\,110\,938\,409\,751\,724\,98$$

$$fl(b) = 1,111\,111\,111\,111\,110\,050\,231\,332\,024\,85$$

$$fl(a) - fl(b) = 8,881\,784\,197\,001\,300 \times 10^{-16}$$

$$fl(fl(a) - fl(b)) = 8,881\,784\,197\,001\,252 \times 10^{-16}$$

Abbiamo così ricostruito l'output restituito da Matlab, che è frutto di una propagazione dell'errore relativa alla rappresentazione dei numeri decimali in macchina.

## 2 Capitolo 2

### 2.1 Esercizio 5

Scrivere *function* Matlab distinte che implementino efficientemente i seguenti metodi per la ricerca degli zeri di una funzione:

- Metodo di bisezione;
- Metodo di Newton;
- Metodo delle secanti;
- Metodo delle corde.

Detta  $x_i$ , l'approssimazione al passo  $i$ -esimo, utilizzare come criterio di arresto

$$|\Delta x_i| \leq tol \cdot (1 + |x_i|),$$

essendo  $tol$  una opportuna tolleranza specificata in ingresso.

---

**Metodo di bisezione:**

```
1 function [x, i, imax] = bisection(f, a, b, tol)
2 %
3 % [x, i, imax] = bisection(f, a, b, tol) Determina uno zero della funzione in ingresso,
4 %                                     sull'intervallo [a, b],
5 %                                     utilizzando il metodo di bisezione.
6 %
7 fa = feval(f, a);
8 fb = feval(f, b);
9 if fa == 0
10     x = a;
11     return
12 end
13 if fb == 0
14     x = b;
15     return
16 end
17 if abs(fa) == Inf || abs(fb) == Inf
18     error('Funzione non valutabile in uno degli estremi.')
19     return
20 end
21 if fa * fb > 0
22     error('Funzione non soddisfacente la condizione f(a) * f(b) < 0.')
23     return
24 end
25
26 x = (a + b) / 2;
27 fx = feval(f, x);
28 imax = ceil(log2(b - a) - log2(tol * (1 + abs(x))));
29 for i = 2 : imax
30     flx = ((fb - fa) / (b - a));
31     if abs(fx / flx) <= tol * (1 + abs(x))
32         break
33     elseif fa * fx < 0
34         b = x;
35         fb = fx;
```

```

36     else
37         a = x;
38         fa = fx;
39     end
40     x = (a + b) / 2;
41     fx = feval(f, x);
42     imax = ceil(log2(b - a) - log2(tol * (1 + abs(x))));
43 end
44 return

```

#### Metodo di Newton:

```

1 function [x, i] = newton(f, f1, x0, imax, tol)
2 %
3 % [x, i] = newton(f, f1, x0, imax, tol) Determina uno zero della funzione
4 %                                     in ingresso utilizzando il metodo di Newton.
5 %
6 fx = feval(f, x0);
7 f1x = feval(f1, x0);
8 x = x0 - fx / f1x;
9 i = 0;
10 go = 1;
11 while go && i < imax
12     i = i + 1;
13     x0 = x;
14     fx = feval(f, x0);
15     f1x = feval(f1, x0);
16     if f1x == 0, error('La derivata prima ha assunto valore zero, impossibile continuare.
17         '), end
18     x = x0 - fx / f1x;
19     go = abs(x - x0) > tol * (1 + abs(x));
20 end
21 if go, disp('Il metodo non converge.'), end
22 return

```

#### Metodo delle secanti:

```

1 function [x, i] = secant(f, f1, x0, imax, tol)
2 %
3 % [x, i] = secant(f, f1, x0, imax, tol) Determina uno zero della funzione
4 %                                     in ingresso utilizzando il
5 %                                     metodo delle secanti.
6 %
7 fx = feval(f, x0);
8 f1x = feval(f1, x0);
9 if f1x == 0, error('La derivata prima ha assunto valore zero, impossibile continuare.'),
10     end
11 x = x0 - fx / f1x;
12 i = 0;
13 go = 1;
14 while go && i < imax
15     i = i + 1;
16     fx0 = fx;
17     fx = feval(f, x);
18     t = (fx - fx0);
19     if t == 0, error('Impossibile determinare la radice nella tolleranza desiderata.'),
20         end
21     x1 = (fx * x0 - fx0 * x) / t;
22     x0 = x;

```



```

21     x = x1;
22     go = abs(x - x0) > tol * (1 + abs(x));
23 end
24 if go, disp('Il metodo non converge.'), end
25 return

```

Metodo delle corde:

```

1 function [x, i] = chord(f, f1, x, imax, tol)
2 %
3 % [x, i] = chord(f, f1, x, imax, tol) Determina uno zero della funzione
4 %                                     in ingresso utilizzando il
5 %                                     metodo delle corde.
6 %
7 flx = feval(f1, x);
8 if flx == 0, error('La derivata prima ha valore nullo, impossibile continuare.'), end
9 go = 1;
10 i = 0;
11 while go && i < imax
12     i = i + 1;
13     x0 = x;
14     fx = feval(f, x0);
15     x = x0 - fx / flx;
16     go = abs(x - x0) > tol * (1 + abs(x));
17 end
18 if go, disp('Il metodo non converge.'), end
19 return

```

## 2.2 Esercizio 6

Utilizzare la *function* del precedente esercizio per determinare un'approssimazione della radice della funzione

$$f(x) = x - e^{-x} \cos\left(\frac{x}{100}\right)$$

per  $tol = 10^{-i}, i = 1, 2, \dots, 12$ , partendo da  $x_0 = -1$ . Per il metodo di bisezione utilizzare  $[-1, 1]$ , come intervallo di confidenza iniziale. Tabulare i risultati, in modo da confrontare le iterazioni richieste da ciascun metodo. Commentare il relativo costo computazionale.

### Codice Matlab

```

1 f = @(x) x - exp(-x) * cos(x / 100);
2 f1 = @(x) 1 + exp(-x) * cos(x / 100) + (exp(-x) * sin(x / 100)) / 100;
3 x0 = -1;
4 a = -1;
5 b = 1;
6 imax = 1000;
7 for i = 1 : 12
8     tol = 10^(-i);
9     rtol=[ 'Tolleranza: ', num2str(tol, '%e')];
10    disp(rtol)
11    [xB, it] = bisection(f, a, b, tol);
12    disp(['Bisezione: ', num2str(xB, '%10.12e'), ' Iterazioni ', num2str(it)])
13    [xN, it] = newton(f, f1, x0, imax, tol);
14    disp(['Newton: ', num2str(xN, '%10.12e'), ' Iterazioni ', num2str(it)])
15    [xC, it] = chord(f, f1, x0, imax, tol);
16    disp(['Corde: ', num2str(xC, '%10.12e'), ' Iterazioni ', num2str(it)])
17    [xS, it] = secant(f, f1, x0, imax, tol);
18    disp(['Secanti: ', num2str(xS, '%10.12e'), ' Iterazioni ', num2str(it)])
19    disp(' ')
20 end

```

### Risultati

Tolleranza	Bisezione		Newton		Secanti		Corde	
	Risultato	Iterazioni	Risultato	Iterazioni	Risultato	Iterazioni	Risultato	Iterazioni
$10^{-1}$	5.000000000000e-01	3	5.663058026183e-01	2	5.662928457961e-01	3	4.021808606807e-01	3
$10^{-2}$	5.625000000000e-01	6	5.671373451066e-01	3	5.671339926161e-01	4	5.495185718942e-01	7
$10^{-3}$	5.664062500000e-01	10	5.671373451066e-01	3	5.671339926161e-01	4	5.651741531555e-01	11
$10^{-4}$	5.671386718750e-01	14	5.671374702932e-01	4	5.671374697616e-01	5	5.670103627779e-01	16
$10^{-5}$	5.671386718750e-01	14	5.671374702932e-01	4	5.671374697616e-01	5	5.671232371728e-01	20
$10^{-6}$	5.671386718750e-01	14	5.671374702932e-01	4	5.671374702932e-01	6	5.671358764609e-01	24
$10^{-7}$	5.671374797821e-01	24	5.671374702932e-01	4	5.671374702932e-01	6	5.671372918144e-01	28
$10^{-8}$	5.671374797821e-01	24	5.671374702932e-01	5	5.671374702932e-01	6	5.671374503070e-01	32
$10^{-9}$	5.671374704689e-01	31	5.671374702932e-01	5	5.671374702932e-01	6	5.671374689985e-01	37
$10^{-10}$	5.671374702360e-01	34	5.671374702932e-01	5	5.671374702932e-01	7	5.671374701482e-01	41
$10^{-11}$	5.671374702943e-01	36	5.671374702932e-01	5	5.671374702932e-01	7	5.671374702770e-01	45
$10^{-12}$	5.671374702943e-01	36	5.671374702932e-01	5	5.671374702932e-01	7	5.671374702914e-01	49

Analizzando i risultati ottenuti, abbiamo verificato che i metodi di bisezione e corde impiegano un numero considerevolmente maggiore di iterazioni rispetto ai metodi di Newton e secanti, a parità di tolleranza utilizzata. Questo è dato dal fatto che i primi due hanno convergenza lineare, mentre i secondi due hanno convergenza quadratica.

## 2.3 Esercizio 7

Calcolare la molteplicità della radice nulla della funzione

$$f(x) = x^2 \sin(x^2).$$

Confrontare, quindi, i metodi di Newton, Newton modificato, e di Aitken, per approssimarla per gli stessi valori di  $tol$  del precedente esercizio (ed utilizzando il medesimo criterio di arresto), partendo da  $x_0 = 1$ . Tabulare e commentare i risultati ottenuti.

---

Metodo di Newton modificato:

```
1 function [x, i] = modnewton(f, f1, x0, m, imax, tol)
2 %
3 % [x, i] = modnewton(f, f1, x0, m, imax, tol) Determina uno zero della funzione
4 %                                     in ingresso, con molteplicita' multipla,
5 %                                     utilizzando il metodo di Newton modificato.
6 %
7 fx = feval(f, x0);
8 f1x = feval(f1, x0);
9 x = x0 - m * (fx / f1x);
10 i = 0;
11 go = 1;
12 while go && i < imax
13     i = i + 1;
14     x0 = x;
15     fx = feval(f, x0);
16     f1x = feval(f1, x0);
17     if f1x == 0, error('La derivata prima ha assunto valore zero, impossibile continuare.
18         '), end
19     x = x0 - m * (fx / f1x);
20     go = abs(x - x0) > tol * (1 + abs(x));
21 end
22 if go, disp('Il metodo non converge.'), end
23 return
```

Metodo di accelerazione di Aitken:

```
1 function [x, i] = aitken(f, f1, x0, imax, tol)
2 %
3 % [x, i] = aitken(f, f1, x0, imax, tol) Determina uno zero della funzione
4 %                                     in ingresso, con molteplicita' multipla,
5 %                                     utilizzando il metodo di Aitken.
6 %
7 x = x0;
8 i = 0;
9 go = 1;
10 while go && i < imax
11     i = i + 1;
12     x0 = x;
13     fx = feval(f, x0);
14     f1x = feval(f1, x0);
15     if f1x == 0, error('La derivata prima ha assunto valore zero, impossibile continuare.
16         '), end
17     x1 = x0 - fx / f1x;
18     fx = feval(f, x1);
```

```

18     f1x = feval(f1, x1);
19     if f1x == 0
20         if abs(x1 - x0) < tol * (1 + abs(x))
21             x = x1;
22             disp('Nella tolleranza richiesta, ma con approssimazioni intermedie al metodo
                di Aitken.');
```

```

23             return
24         end
25         error('La derivata prima ha assunto valore zero, impossibile continuare.')
```

```

26     end
27     x = x1 - fx / f1x;
28     t = (x - 2 * x1 + x0);
29     if t == 0
30         if abs(x - x1) < tol * (1 + abs(x))
31             disp('Nella tolleranza richiesta, ma con approssimazioni intermedie al metodo
                di Aitken.');
```

```

32             return
33         end
34         error('Impossibile determinare la radice nella tolleranza desiderata.')
```

```

35     end
36     x = (x * x0 - (x1)^2) / t;
37     go = abs(x - x0) > tol * (1 + abs(x));
38 end
39 if go, disp('Il metodo non converge. '), end
40 return

```

Dato che il metodo di Newton modificato richiede espressamente la molteplicità della radice della funzione in esame, andiamo a calcolarla:

$$x^2 \sin(x^2) = 0 \rightarrow x^2 = 0 \vee \sin(x^2) = 0$$

Le radici del polinomio sono 0 e  $k\sqrt{\pi}$ , con  $k \in \mathbb{Z}$ . La molteplicità della radice  $x = 0$  è pari a 3, mentre quella della radice  $x = k\sqrt{\pi}$  è pari a 1. Dunque, nel nostro caso, per l'utilizzo del metodo di Newton modificato, inizializzeremo il valore  $m$  relativo alla molteplicità a 3.

Codice Matlab

```

1  f = @(x) x^2 * sin(x^2);
2  f1 = @(x) 2 * x * (sin(x^2) + x^2 * cos(x^2));
3  x0 = 1;
4  imax = 1000;
5  m = 3;
6  for i = 1 : 12
7      tol = 10^(-i);
8      rtol=[ 'Tolleranza: ', num2str(tol, '%e')];
9      disp(rtol)
10     [xN, it] = newton(f, f1, x0, imax, tol);
11     disp(['Newton: ', num2str(xN, '%10.12e'), ' Iterazioni ', num2str(it)])
12     [xNM, it] = modnewton(f, f1, x0, m, imax, tol);
13     disp(['Newton Modificato: ', num2str(xNM, '%10.12e'), ' Iterazioni ', num2str(it)
14           ])
15     [xA, it] = aitken(f, f1, x0, imax, tol);
16     disp(['Aitken: ', num2str(xA, '%10.12e'), ' Iterazioni ', num2str(it)])
17     disp(' ')
18 end

```

## Risultati

Tolleranza	Newton		Newton modificato		Aitken	
	Risultato	Iterazioni	Risultato	Iterazioni	Risultato	Iterazioni
$10^{-1}$	3.843178806071e-01	2	2.163225010255e-02	1	6.492908618812e-19	3
$10^{-2}$	2.880513930938e-02	11	1.352015482798e-03	3	6.492908618812e-19	3
$10^{-3}$	2.883766303035e-03	19	8.450096767474e-05	5	6.492908618812e-19	3
$10^{-4}$	2.887022508866e-04	27	2.112524191869e-05	6	0.000000000000e+00	4
$10^{-5}$	2.890282391460e-05	35	1.320327619918e-06	8	0.000000000000e+00	4
$10^{-6}$	2.893545954951e-06	43	3.300819049795e-07	9	0.000000000000e+00	4
$10^{-7}$	2.896813203496e-07	51	2.063011906122e-08	11	0.000000000000e+00	4
$10^{-8}$	2.900084141257e-08	59	1.289382441326e-09	13	0.000000000000e+00	4
$10^{-9}$	2.903358772398e-09	67	3.223456103315e-10	14	0.000000000000e+00	4
$10^{-10}$	2.906637101090e-10	75	2.014660064572e-11	16	0.000000000000e+00	4
$10^{-11}$	2.909919131508e-11	83	1.259162540357e-12	18	0.000000000000e+00	4
$10^{-12}$	2.913204867832e-12	91	3.147906350894e-13	19	0.000000000000e+00	4

Analizzando i risultati ottenuti, abbiamo verificato che il metodo di Newton impiega un numero considerevolmente maggiore di iterazioni rispetto ai metodi di Newton modificato e Aitken, a parità di tolleranza utilizzata. Questo è dato dal fatto che il primo ha convergenza lineare (in caso di radici multiple), mentre i secondi due hanno convergenza quadratica.

## 3 Capitolo 3

### 3.1 Esercizio 8

Scrivere una *function* Matlab che, data in ingresso una matrice  $A$ , restituisca una matrice,  $LU$ , che contenga l'informazione sui suoi fattori  $L$  ed  $U$ , ed un vettore  $\mathbf{p}$  contenente la relativa permutazione, della fattorizzazione  $LU$  con *pivoting* parziale di  $A$ :

$$function[LU,p] = palu(A)$$

Curare particolarmente la scrittura e l'efficienza della *function*.

---

Fattorizzazione LU con pivoting parziale:

```
1 function [A, p] = palu (A)
2 %
3 % function [A, p] = palu (A) calcola la fattorizzazione LU con pivoting parziale e
4 % restituisce la matrice fattorizzata e il vettore
5 % contenente l'informazione relativa alla matrice
6 % di permutazione
7 %
8 [m, n] = size(A);
9 if m ~= n
10     error('Matrice non quadrata.');
```

### 3.2 Esercizio 9

Scrivere una *function* Matlab che, data in ingresso la matrice  $LU$  ed il vettore  $\mathbf{p}$  creati dalla *function* del precedente esercizio, ed il termine noto del sistema lineare  $A\mathbf{x} = \mathbf{b}$ , ne calcoli la soluzione:

$$function x = lusolve(LU, p, b)$$

Curare particolarmente la scrittura e l'efficienza della *function*.

---

Risoluzione di un sistema lineare, con matrice dei coefficienti fattorizzata LU con pivoting parziale:

```

1 function x = palusolve(LU, p, b)
2 %
3 % x = palusolve(LU, p, b) risolve il sistema lineare LUx = b,
4 %                     con LU matrice fattorizzata LU
5 %                     con pivoting parziale
6 %
7 x = b;
8 [m, n] = size(LU);
9 if n ~= length(x) || m ~= n, error(Dati inconsistenti.), end
10 x = infsolve(LU, x(p));
11 x = supsolve(LU, x);
12 return
13
14 function [x] = infsolve(L, x)
15 n = length(x);
16 for i = 1:n
17     x(i + 1 : n) = x(i + 1 : n) - L(i + 1 : n, i) * x(i);
18 end
19 return
20
21 function [x] = supsolve(U, x)
22 n = length(x);
23 for i = n : -1 : 1
24     x(i) = x(i) / U(i, i);
25     x(1 : i - 1) = x(1 : i - 1) - U(1 : i - 1, i) * x(i);
26 end
27 return

```

### 3.3 Esercizio 11

Scrivere una *function* Matlab che, data in ingresso una matrice  $A \in \mathbb{R}^{m \times n}$ , con  $m \geq n = \text{rank}(A)$ , restituisca una matrice,  $QR$ , che contenga l'informazione sui fattori  $Q$  ed  $R$  della fattorizzazione  $QR$  di  $A$ :

$$\text{function } QR = \text{myqr}(A)$$

Curare particolarmente la scrittura e l'efficienza della *function*.

Fattorizzazione QR:

```

1 function QR = myqr(A)
2 %
3 % QR = myqr(A) calcola la fattorizzazione QR della
4 %             matrice sovradeterminata A
5 %
6 [m, n] = size(A);
7 if m < n, error('Numero di righe della matrice minore del numero di colonne.');
```

```

14     QR(i, i) = alfa;
15     QR(i + 1 : m, i) = QR(i + 1 : m, i) / v1;
16     beta = v1 / alfa;
17     v = [1; QR(i + 1 : m, i)];
18     QR(i : m, i + 1 : n) = QR(i : m, i + 1 : n) + (beta * v) * (v' * QR(i : m, i + 1 : n))
        ;
19 end
20 return

```

### 3.4 Esercizio 12

Scrivere una *function* Matlab che, data in ingresso la matrice  $QR$  creata dalla *function* del precedente esercizio, ed il termine noto del sistema lineare  $A\mathbf{x} = \mathbf{b}$ , ne calcoli la soluzione nel senso dei minimi quadrati:

$$\text{function } x = \text{qrsolve}(QR, b)$$

Curare particolarmente la scrittura e l'efficienza della *function*.

---

**Risoluzione di un sistema lineare, con matrice dei coefficienti fattorizzata QR:**

```

1 function x = qrsolve(QR, b)
2 %
3 % x = qrsolve(QR, b) Risolve sistemi lineari data la matrice QR
4 %             contenente le parti significative di una fattorizzazione
5 %             QR di Householder, che computa g = Q.'b e in seguito Rx = g
6 %
7 B = QR;
8 [m, n] = size(B);
9 if m < n || m ~= length(b), error('Sistema non risolvibile.');
```



## 4 Capitolo 4

### 4.1 Esercizio 14

Scrivere un programma che implementi efficientemente il calcolo del polinomio interpolante su un insieme di ascisse distinte.

---

**Polinomio interpolante di Lagrange:**

```
1 function y = lagrange(xi, fi, x)
2 %
3 % y = lagrange(xi, fi, x) calcola il polinomio interpolante le coppie (xi, fi)
4 %                         sulle ascisse x
5 %
6 n = length(xi); if length(fi) ~= n, error(Dati inconsistenti.), end
7 for i = 1 : n - 1
8     for j = i + 1 : n
9         if xi(i) == xi(j), error(Ascisse non distinte.), end
10    end
11 end
12 y = zeros(size(x));
13 for i = 1 : n
14     if fi(i) ~= 0
15         li = 1;
16         for k = [1 : i - 1 i + 1 : n]
17             li = li .* (x - xi(k)) / (xi(i) - xi(k));
18         end
19         y = y + fi(i) * li;
20     end
21 end
22 return
```

**Polinomio interpolante di Newton:**

```
1 function y = newton(xi, fi, x)
2 %
3 % y = newton(xi, fi, x) calcola il polinomio interpolante le coppie (xi, fi)
4 %                         sulle ascisse x
5 %
6 n = length(xi); if length(fi) ~= n, error(Dati inconsistenti.), end
7 for i = 1 : n - 1
8     for j = i + 1 : n
9         if xi(i) == xi(j), error(Ascisse non distinte.), end
10    end
11 end
12 fi = dividif(xi, fi);
13 y = fi(n) * ones(size(x));
14 for i = n - 1 : -1 : 1
15     y = y .* (x - xi(i)) + fi(i);
16 end
17 return
18
19 function fi = dividif(xi, fi)
20 %
21 % f = dividif(xi, fi) calcola le differenze divise
```

```

22 %           sulle coppie (xi, fi)
23 %
24 n = length(xi);
25 for i = 1 : n - 1
26     for j = n : -1 : i + 1
27         fi(j) = (fi(j) - fi(j - 1)) / (xi(j) - xi(j - i));
28     end
29 end
30 return

```

## 4.2 Esercizio 15

Scrivere un programma che implementi efficientemente il calcolo del polinomio interpolante di Hermite su un insieme di ascisse distinte.

### Polinomio interpolante di Hermite:

```

1 function y = hermite(xi, fi, fli, x)
2 %
3 % y = hermite(xi, fi, fli, x) Calcola il polinomio interpolante di grado n
4 %           in forma di Hermite, nei punti x
5 %
6 n = length(xi) - 1;
7 xh = zeros(2 * n + 2, 1);
8 xh(1 : 2 : 2 * n + 1) = xi;
9 xh(2 : 2 : 2 * n + 2) = fi;
10 fh = zeros(2 * n + 2, 1);
11 fh(1 : 2 : 2 * n + 1) = fi;
12 fh(2 : 2 : 2 * n + 2) = fli;
13 fh = dividif(xh, fh);
14 y = horner(xh, fh, x);
15 return
16
17 function fh = dividif(xh, fh)
18 %
19 % fh = dividif(xh, fh) Calcola le differenze divise sulle coppie (xh, fh)
20 %
21 %
22 nh = length(xh) - 1;
23 for i = nh : -2 : 3
24     fh(i) = (fh(i) - fh(i - 2)) / (xh(i) - xh(i - 1));
25 end
26 for i = 2 : nh
27     for j = nh + 1 : -1 : i + 1
28         fh(j) = (fh(j) - fh(j - 1)) / (xh(j) - xh(j - i));
29     end
30 end
31 return
32
33 function y = horner(xh, fh, x)
34 %
35 % y = horner(xh, fh, x) Valuta il polinomio interpolante nelle ascisse xh,
36 %           se il vettore fh contiene i coefficienti di p,
37 %           ordinati a partire dal termine noto

```

```

38 %
39 nh = length(xh) - 1;
40 y = fh(nh + 1) * ones(size(x));
41 for i = nh : -1 : 1
42     y = y .* (x - xh(i)) + fh(i);
43 end
44 return

```

### 4.3 Esercizio 16

Scrivere un programma che implementi efficientemente il calcolo di una spline cubica naturale interpolante su una partizione assegnata.

**Risoluzione di un sistema tridiagonale:**

```

1 function x = trid(a, b, c, f)
2 %
3 % x = trid(a, b, c, f) risolve il sistema tridiagonale con
4 %             diagonale principale a, sotto-diagonale b
5 %             e sopra-diagonale c, con vettore dei termini noti f
6 %
7 n = length(a);
8 if n ~= length(f) || length(b) >= n || length(c) >= n, error('Dati inconsistenti. '), end
9 d = a;
10 l = b;
11 x = f;
12 for i = 1 : n - 1
13     if d(i) == 0, error('Matrice non fattorizzabile LU. '), end
14     l(i) = b(i) / d(i);
15     x(i + 1) = x(i + 1) - l(i) * x(i);
16     d(i + 1) = d(i + 1) - l(i) * c(i);
17 end
18 if d(n) == 0, error('Matrice non fattorizzabile LU. '), end
19 x(n) = x(n) / d(n);
20 for i = n - 1 : -1 : 1
21     x(i) = (x(i) - c(i) * x(i + 1)) / d(i);
22 end
23 return

```

**Spline cubica naturale:**

```

1 function y = spline3(xi, fi, x)
2 %
3 % y = spline3(xi, fi, x) Calcola la spline cubica naturale
4 %             su una partizione assegnata e
5 %             la valuta nelle ascisse date
6 %
7 if any(size(xi) ~= size(fi)) || size(xi, 2) ~= 1
8     error('xi e fi devono essere vettori della stessa lunghezza. ');
9 end
10 if length(xi) < 3, error('Sono necessari almeno 3 punti. '), end
11 [xi, p] = sort(xi);
12 fi = fi(p);
13

```

```

14 h = xi(2 : end) - xi(1 : end - 1);
15 n = length(h);
16
17 phi = h(1 : n - 1) ./ (h(1 : n - 1) + h(2 : n));
18 csi = 1 - phi;
19
20 d = (fi(2 : end) - fi(1 : end - 1)) ./ h;
21 b = 6 * ((d(2 : n) - d(1 : n - 1)) ./ (h(1 : n - 1) + h(2 : n)));
22
23 lower = phi(2 : n - 1);
24 main = 2 * ones(n - 1, 1);
25 upper = csi(1 : n - 2);
26
27 m = [0; trid(main, lower, upper, b); 0];
28
29 ri = fi(1 : end - 1) - (h.^2 .* m(1 : end - 1)) / 6;
30 qi = d - (h .* (m(2 : end) - m(1 : end - 1))) / 6;
31
32 l = length(x);
33 y = zeros(l, 1);
34 for j = 1 : l
35     if x(j) < xi(1) || x(j) > xi(end)
36         error('Funzione interpolante non valutabile nelle ascisse date.')
37     end
38     i = 1;
39     while i < n + 1 && xi(i) <= x(j), i = i + 1; end
40     y(j) = ( (x(j) - xi(i - 1))^3 * m(i) + (xi(i) - x(j))^3 * m(i - 1) ) / (6 * h(i - 1))
41         ;
42     y(j) = y(j) + qi(i - 1) * (x(j) - xi(i - 1)) + ri(i - 1);
43 end
44 return

```

#### 4.4 Esercizio 17

Scrivere un programma che implementi il calcolo di una spline cubica not-a-knot interpolante su una partizione assegnata.

**Spline cubica naturale:**

```

1 function y = spline3nak(xi, fi, x)
2 %
3 % y = spline3nak(xi, fi, x) Calcola la spline cubica not-a-knot
4 %                           su una partizione assegnata e
5 %                           la valuta nelle ascisse date
6 %
7 if any(size(xi) ~= size(fi)) || size(xi, 2) ~= 1
8     error('xi e fi devono essere vettori della stessa lunghezza.');
```

```

16
17 phi = h(1 : n - 1) ./ (h(1 : n - 1) + h(2 : n));
18 csi = 1 - phi;
19
20 lower = [phi(1 : n - 2); phi(n - 1) - csi(n - 1); 0];
21 main = [1; 2 - phi(1); 2 * ones(n - 3, 1); 2 - csi(n - 1); 1];
22 upper = [0; csi(1) - phi(1); csi(2 : n - 1)];
23
24 d = (fi(2 : end) - fi(1 : end - 1)) ./ h;
25 b = 6 * ((d(2 : n) - d(1 : n - 1)) ./ (h(1 : n - 1) + h(2 : n)));
26 b = [b(1); b; b(end)];
27
28 m = trid(main, lower, upper, b);
29
30 m(1) = m(1) - m(2) - m(3);
31 m(end) = m(end) - m(end - 1) - m(end - 2);
32
33 ri = fi(1 : end - 1) - (h.^2 .* m(1 : end - 1)) / 6;
34 qi = d - (h .* (m(2 : end) - m(1 : end - 1))) / 6;
35
36 l = length(x);
37 y = zeros(l, 1);
38 for j = 1 : l
39     if x(j) < xi(1) || x(j) > xi(end)
40         error('Funzione interpolante non valutabile nelle ascisse date.')
41     end
42     i = 1;
43     while i < n + 1 && xi(i) <= x(j), i = i + 1; end
44     y(j) = ( (x(j) - xi(i - 1))^3 * m(i) + (xi(i) - x(j))^3 * m(i - 1) ) / (6 * h(i - 1))
45         ;
46     y(j) = y(j) + qi(i - 1) * (x(j) - xi(i - 1)) + ri(i - 1);
47 end
48 return

```

## 4.5 Esercizio 18

Confrontare i codici degli esercizi 14-17 per approssimare la funzione  $f(x) = \sin(x)$  sulle ascisse  $x_i = i\pi/n, i = 0, 1, \dots, n$ , per  $n = 1, 2, \dots, 10$ . Graficare l'errore massimo di approssimazione verso  $n$  (in semilog), calcolato su una griglia uniforme di 10001 punti nell'intervallo  $[0, \pi]$ .

### Codice Matlab

```

1 f = @(x) sin(x);
2 f1 = @(x) cos(x);
3 x = linspace(0, pi, 10001)';
4 fx = f(x);
5
6 el = zeros(10, 1); en = zeros(10, 1); eh = zeros(10, 1);
7 esn = zeros(10, 1); esnak = zeros(10, 1);
8 for n = 1 : 10
9     xi = zeros(n + 1, 1);
10    for i = 0 : n, xi(i + 1) = (i * pi) / n; end
11    fi = f(xi);
12    fli = f1(xi);

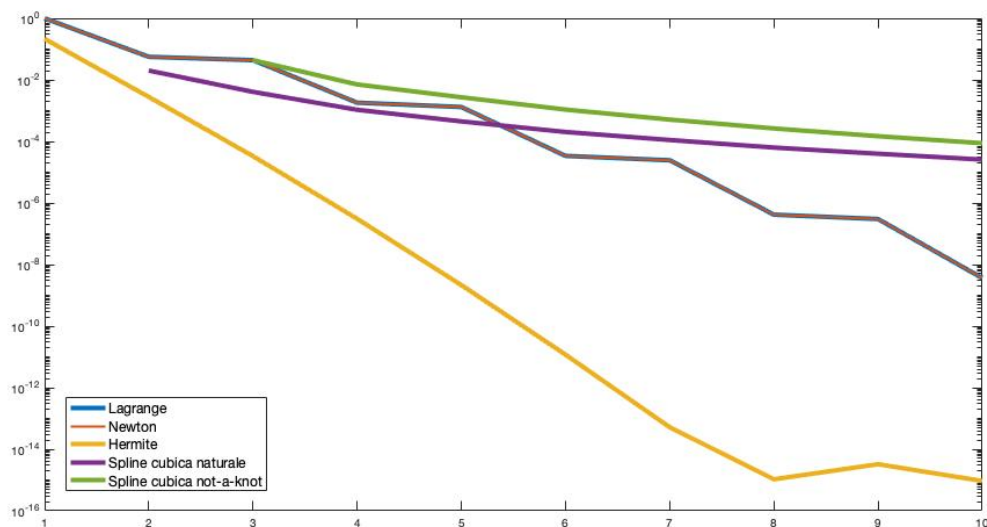
```

```

13
14     yl = lagrange(xi, fi, x);
15     el(n) = norm(fx - yl, inf);
16
17     yn = newton(xi, fi, x);
18     en(n) = norm(fx - yn, inf);
19
20     yh = hermite(xi, fi, fli, x);
21     eh(n) = norm(fx - yh, inf);
22
23     if n >= 2
24         ysn = spline3(xi, fi, x);
25         esn(n) = norm(fx - ysn, inf);
26     end
27
28     if n >= 3
29         ysnak = spline3nak(xi, fi, x);
30         esnak(n) = norm(fx - ysnak, inf);
31     end
32
33 end
34 x = 1 : 10;
35 semilogy(x, el, x, en, x, eh, x, esn, x, esnak);
36 legend('Lagrange', 'Newton', 'Hermite', 'Spline cubica naturale', 'Spline cubica not-a-
    knot');

```

La seguente figura mostra gli errori assoluti massimi dei vari metodi di interpolazione implementati:



Errori assoluti commessi per approssimare la funzione con  $n = 1 : 10$

## 4.6 Esercizio 19

Calcolare (numericamente) la costante di Lebesgue per i polinomi interpolanti di grado  $n = 2, 4, 8, \dots, 40$ , sia sulle ascisse equidistanti che su quelle di Chebyshev (utilizzare 10001 punti equispaziati per valutare la funzione di Lebesgue). Graficare convenientemente i risultati ottenuti. Spiegare, quindi, i risultati ottenuti approssimando la funzione

$$f(x) = \frac{1}{1+x^2}, x \in [-5, 5]$$

utilizzando le ascisse equidistanti e di Chebyshev precedentemente menzionate (tabulare il massimo errore valutato su una griglia di 10001 punti equidistanti nell'intervallo  $[-5, 5]$ ).

**Costante di Lebesgue:**

```

1 function leb = lebesgue(x)
2 %
3 % leb = lebesgue(x) Calcola il valore della costante di Lebesgue,
4 % dato il vettore x di N nodi (ordinato)
5 %
6 n = length(x);
7 x_g = linspace(x(1), x(end), 10001);
8 m = length(x_g);
9 vLeb = zeros(m, 1);
10 for i = 1 : m
11     for j = 1 : n
12         range = [1 : j - 1, j + 1 : n];
13         bl = prod(x_g(i) - x(range)) / prod(x(j) - x(range));
14         vLeb(i) = vLeb(i) + abs(bl);
15     end
16 end
17 leb = norm(vLeb, inf);
18 end

```

**Ascisse di Chebyshev:**

```

1 function xi = ceby(n, a, b)
2 %
3 % xi = ceby(n, a, b) Calcola le ascisse di Chebyshev
4 % sull'intervallo [a, b]
5 %
6 xi = zeros(n + 1, 1);
7 for i = 0 : n
8     xi(n + 1 - i) = (a + b) / 2 + cos(pi * (2 * i + 1) / (2 * (n + 1))) * (b - a) / 2;
9 end
10 end

```

Codice Matlab

```

1 f = @(x) 1 ./ (1 + x.^2);
2 a = -5;
3 b = 5;
4 n = 2 : 2 : 40;
5 kxi = zeros(length(n), 1);
6 kci = zeros(length(n), 1);
7 x = linspace(a, b, 10001)';
8 fx = f(x);
9 ex = zeros(length(n), 1); ec = zeros(length(n), 1);
10
11 figure(1)
12 ax1 = subplot(2, 1, 1);
13 for i = 1 : length(n)
14     xi = linspace(a, b, n(i) + 1);
15     fxi = f(xi);
16     yxi = lagrange(xi, fxi, x);
17     plot(ax1, x, yxi)

```

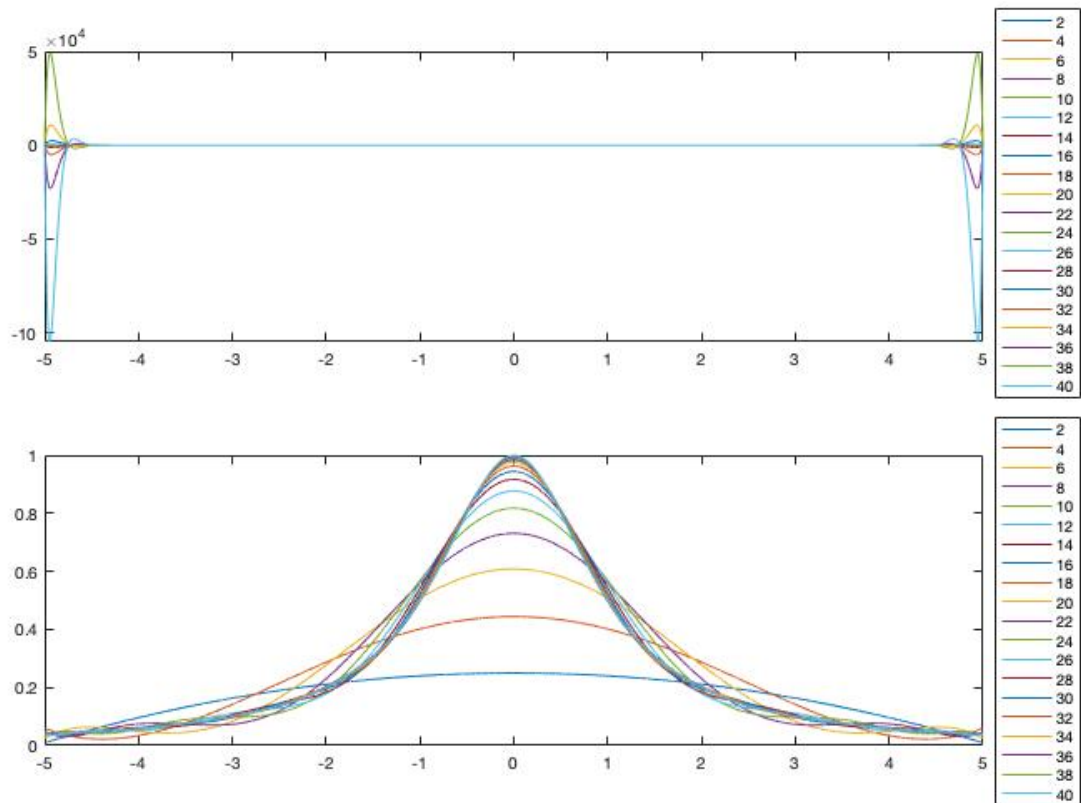
```

18         hold on
19         kxi(i, 1) = lebesgue(xi);
20         ex(i) = norm(fx - yxi, inf);
21     end
22     legend('2', '4', '6', '8', '10', '12', '14', '16', '18', '20', '22', '24', '26', '28', '30', '32', '34', '36', '38', '40')
23     hold off
24
25     ax2 = subplot(2, 1, 2);
26     for i = 1 : length(n)
27         ci = ceby(n(i) + 1, a, b);
28         x = linspace(min(ci), max(ci), 10001)';
29         fx = f(x);
30         fci = f(ci);
31         yci = lagrange(ci, fci, x);
32         plot(ax2, x, yci)
33         hold on
34         kci(i, 1) = lebesgue(ci);
35         ec(i) = norm(fx - yci, inf);
36     end
37     legend('2', '4', '6', '8', '10', '12', '14', '16', '18', '20', '22', '24', '26', '28', '30', '32', '34', '36', '38', '40')
38     hold off
39
40     figure(2)
41     plot(n, kxi, n, kci);
42     legend('Ascisse equispaziate', 'Ascisse di Chebyshev')

```

Le seguenti figure mostrano il polinomio di *Lagrange*, al variare del grado  $n$  del polinomio con  $n = 2, 4, 6, 8, \dots, 40$ , utilizzando ascisse equidistanti e ascisse di Chebyshev:





Polinomio di Lagrange con grado  $n = 2, 4, 6, 8 \dots 40$ , interpolante ascisse equidistanti e di Chebyshev, rispettivamente

Nelle seguenti tabelle è riportato come varia la *costante di Lebesgue*  $\Lambda$ , al variare del grado  $n$  del polinomio e si può notare come la crescita sia *esponenziale*, per  $n \rightarrow \infty$ , prendendo in considerazione *ascisse equidistanti*:

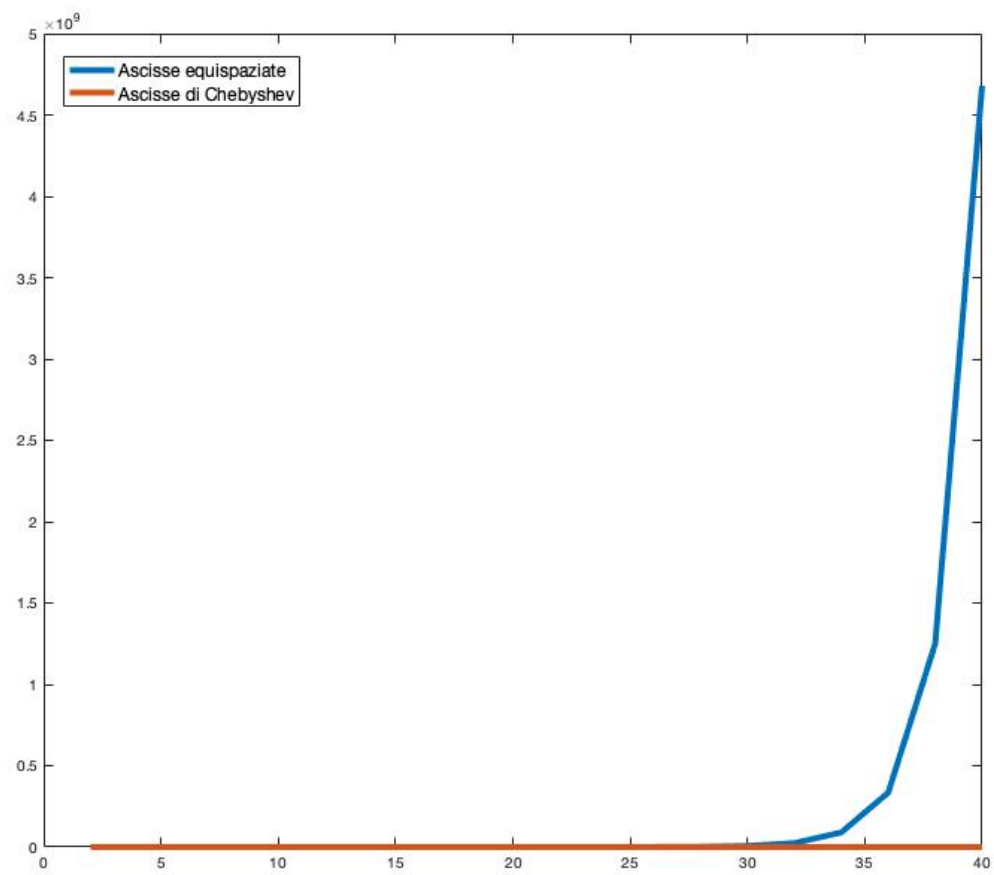
Costante di Lebesgue con ascisse equispaziate

$n$	$\Lambda$
2	1.250000000000000
4	2.207824277504000
6	4.549341110838356
8	10.945005461386041
10	29.898141093562188
12	89.323735973507041
14	$2.831809493441890e + 02$
16	$9.342736404823136e + 02$
18	$3.170339307979169e + 03$
20	$1.097924392398584e + 04$
22	$3.866684343844037e + 04$
24	$1.378514896760509e + 05$
26	$4.964824917524024e + 05$
28	$1.802445465492321e + 06$
30	$6.592504744423425e + 06$
32	$2.430870357380395e + 07$
34	$8.978560703086898e + 07$
36	$3.348225693891219e + 08$
38	$1.249687039228850e + 09$
40	$4.678649708006595e + 09$

Costante di Lebesgue con ascisse di Chebyshev

$n$	$\Lambda$
2	$1.429872254730518e + 00$
4	$1.685135237733246e + 00$
6	$1.866863755668355e + 00$
8	$2.008324271512492e + 00$
10	$2.123677735937467e + 00$
12	$2.221698051371436e + 00$
14	$2.304741888790266e + 00$
16	$2.381233643402699e + 00$
18	$2.447573290086530e + 00$
20	$2.508706712856935e + 00$
22	$2.549521672833381e + 00$
24	$2.612618302290017e + 00$
26	$2.647525891619366e + 00$
28	$2.699620139181539e + 00$
30	$2.742631723381937e + 00$
32	$2.771163414194285e + 00$
34	$2.802963771799375e + 00$
36	$2.839252073343268e + 00$
38	$2.869838999732480e + 00$
40	$2.909700595498265e + 00$

La seguente figura mostra in forma grafica i precedenti dati, espressi in forma tabellare:



Crescita della costante di Lebesgue

Nelle seguenti tabelle viene invece riportato come varia il massimo errore di approssimazione, al variare del grado  $n$  del polinomio:

Massimo errore con ascisse equispaziate

$n$	<i>Errore</i>
2	$6.462292487487394e - 01$
4	$4.383571218947540e - 01$
6	$6.169479236760332e - 01$
8	$1.045176501871859e + 00$
10	$1.915658802784827e + 00$
12	$3.663392805417856e + 00$
14	$7.194881107233104e + 00$
16	$1.439385128500340e + 01$
18	$2.919043772698086e + 01$
20	$5.982230871072865e + 01$
22	$1.236242551524645e + 02$
24	$2.572129123346204e + 02$
26	$5.381745497459237e + 02$
28	$1.131420473604506e + 03$
30	$2.388280971350734e + 03$
32	$5.058959841094824e + 03$
34	$1.074904570555982e + 04$
36	$2.290122855213777e + 04$
38	$4.890718541503938e + 04$
40	$1.046676859396163e + 05$

Massimo errore con ascisse di Chebyshev

$n$	<i>Errore</i>
2	$7.503001200480193e - 01$
4	$5.559113388123953e - 01$
6	$3.917402845902284e - 01$
8	$2.691783353450816e - 01$
10	$1.827582819736332e - 01$
12	$1.233976719701794e - 01$
14	$8.310704778474631e - 02$
16	$5.590747791183015e - 02$
18	$3.759032889290592e - 02$
20	$2.526854568700831e - 02$
22	$1.698393147777255e - 02$
24	$1.141498554193532e - 02$
26	$7.671902699208921e - 03$
28	$5.156161990762298e - 03$
30	$3.465357930583779e - 03$
32	$2.328996182090259e - 03$
34	$1.565269282134074e - 03$
36	$1.051984088923263e - 03$
38	$7.070159314990221e - 04$
40	$4.751701960865606e - 04$

Dunque, possiamo osservare che avviene il cosiddetto **fenomeno di Runge**, ovvero un problema relativo all'interpolazione polinomiale su nodi equispaziati con polinomi di grado elevato. È possibile provare che in tal caso l'errore di approssimazione tende all'infinito all'aumentare del grado del polinomio, anche osservando i risultati ottenuti. Per evitare tale fenomeno, è necessario ricorrere all'utilizzo dei nodi di Chebyshev, oppure all'interpolazione tramite funzioni spline.

#### 4.7 Esercizio 20

Con riferimento al precedente esercizio, tabulare il massimo errore di approssimazione (calcolato come sopra indicato), sia utilizzando le ascisse equidistanti che quelle di Chebyshev su menzionate, relativo alla spline cubica naturale interpolante  $f(x)$  su tali ascisse.

#### Codice Matlab

```

1 f = @(x) 1 ./ (1 + x.^2);
2 a = -5;
3 b = 5;
4 n = 2 : 2 : 40;
5 x = linspace(a, b, 10001)';
6 fx = f(x);
7 ex = zeros(length(n), 1); ec = zeros(length(n), 1);
8
9 ax1 = subplot(2, 1, 1);
10 for i = 1 : length(n)
11     xi = linspace(a, b, n(i) + 1)';
12     fxi = f(xi);
13     yxi = spline3(xi, fxi, x);
14     plot(ax1, x, yxi)
15     hold on
16     ex(i) = norm(fx - yxi, inf);

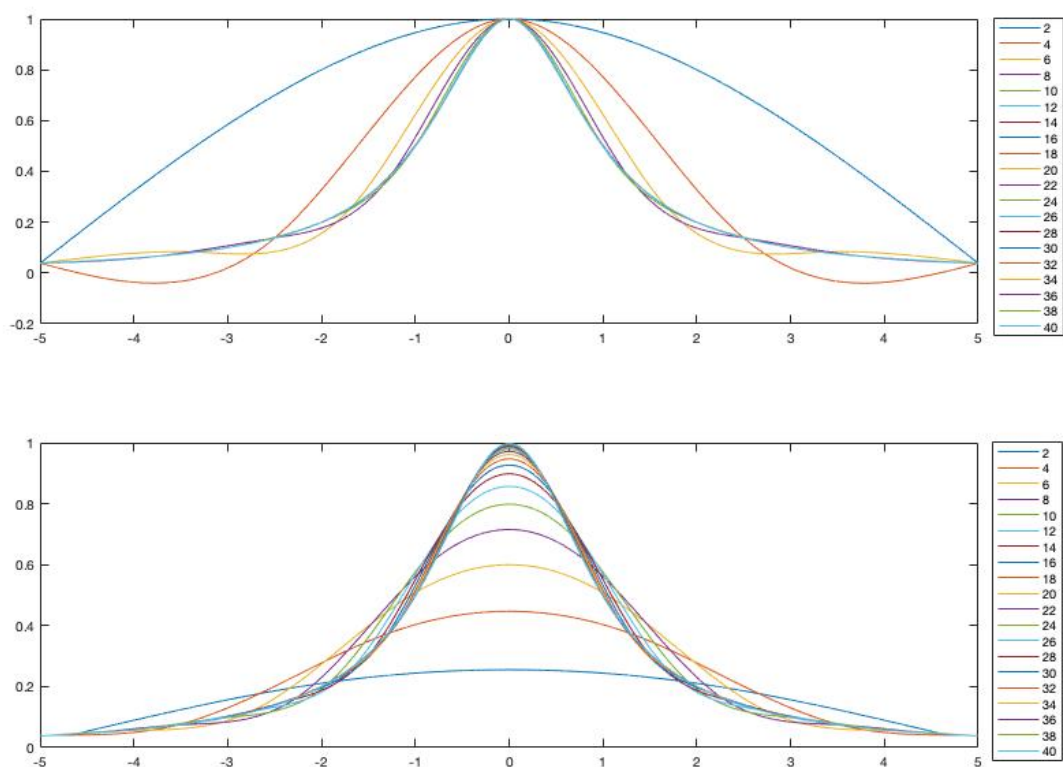
```

```

17 end
18 legend('2', '4', '6', '8', '10', '12', '14', '16', '18', '20', '22', '24', '26', '28', '30', '32', '34', '36', '38', '40')
19 hold off
20
21 ax2 = subplot(2, 1, 2);
22 for i = 1 : length(n)
23     ci = ceby(n(i) + 1, a, b);
24     x = linspace(min(ci), max(ci), 10001)';
25     fx = f(x);
26     fci = f(ci);
27     yci = spline3(ci, fci, x);
28     plot(ax2, x, yci)
29     hold on
30     ec(i) = norm(fx - yci, inf);
31 end
32 legend('2', '4', '6', '8', '10', '12', '14', '16', '18', '20', '22', '24', '26', '28', '30', '32', '34', '36', '38', '40')
33 hold off

```

Le seguenti figure mostrano la spline cubica naturale interpolante  $f(x)$ , al variare del numero di ascisse di interpolazione ( $= 3, 5, 7, 9, \dots, 41$ ), utilizzando ascisse equidistanti e ascisse di Chebyshev:



Spline cubica naturale con  $n^\circ$  di ascisse  $[3, 5, 7, 9, \dots, 41]$ , interpolante ascisse equidistanti e di Chebyshev, rispettivamente

Nelle seguenti tabelle è riportato come varia il massimo errore di approssimazione, al variare del numero di ascisse di interpolazione:

Massimo errore con ascisse equispaziate

$n$	<i>Errore</i>
2	$6.011945468114989e-01$
4	$2.793134075196790e-01$
6	$1.293000883540983e-01$
8	$5.607385287856159e-02$
10	$2.197382574958184e-02$
12	$6.908801437725876e-03$
14	$2.482863475717023e-03$
16	$3.745402833395861e-03$
18	$3.717998718041349e-03$
20	$3.182857643173609e-03$
22	$2.529653088972128e-03$
24	$1.925792361618828e-03$
26	$1.427047863662545e-03$
28	$1.039053280856961e-03$
30	$8.243623332672145e-04$
32	$6.554986812412622e-04$
34	$5.237082286350114e-04$
36	$4.210035707792326e-04$
38	$3.408377962142994e-04$
40	$2.779765405962475e-04$

Massimo errore con ascisse di Chebyshev

$n$	<i>Errore</i>
2	$7.446676877937543e-01$
4	$5.529300734832576e-01$
6	$4.003323237631188e-01$
8	$2.848203213309948e-01$
10	$2.017549216803890e-01$
12	$1.432128531581862e-01$
14	$1.022002088148625e-01$
16	$7.343428357692028e-02$
18	$5.316898347940879e-02$
20	$3.880700384549352e-02$
22	$2.856026101132547e-02$
24	$2.119758438849628e-02$
26	$1.586841915298964e-02$
28	$1.198228990567030e-02$
30	$9.126979380553069e-03$
32	$7.013001529651786e-03$
34	$5.435813807564749e-03$
36	$4.249991716239854e-03$
38	$3.351491356512803e-03$
40	$2.665402637469283e-03$

Dunque, possiamo notare che il **fenomeno di Runge** non è più presente, dato che l'errore di approssimazione non tende più a infinito. Questo risultato è dato dall'utilizzo delle spline come metodo di interpolazione, grazie alle quali è possibile suddividere la partizione dei nodi in più parti, calcolando su ciascun sottointervallo un polinomio interpolante di grado non elevato (in questo caso di grado 3, dato che abbiamo utilizzato una spline cubica).

## 5 Capitolo 5

### 5.1 Esercizio 22

Scrivere due functions che implementino efficientemente le formule adattative dei trapezi e di Simpson.

Formule adattative dei trapezi:

```
1 function I2 = adaptrap(f, a, b, tol, fa, fb)
2 %
3 % I2 = adaptrap(f, a, b[, tol]) calcola l'integrale definito di f
4 %                               in [a, b], utilizzando le formule
5 %                               adattative dei trapezi
6 %
7 global pt;
8 pt = pt + 1;
9 I2 = 0;
10 if a == b, return, end
11 if b < a
12     t = a;
13     a = b;
14     b = t;
15 end
16 if nargin <= 4
17     if nargin <= 3
18         tol = 1E-6;
19     elseif tol <= 0 || tol >= 0.1
20         error('Tolleranza inconsistente.');
```

Formule adattative di Simpson:

```
1 function I4 = adapsimp(f, a, b, tol, fa, fb, f1)
2 %
3 % I4 = adapsimp(f, a, b[, tol]) calcola l'integrale definito di f
4 %                               in [a, b], utilizzando le formule
5 %                               adattative di Simpson
6 %
7 global ps;
8 ps = ps + 2;
9 I4 = 0;
```

```

10 if a == b, return, end
11 if b < a
12     t = a;
13     a = b;
14     b = t;
15 end
16 x1 = (a + b) / 2;
17 if nargin <= 4
18     if nargin <= 3
19         tol = 1E-6;
20     elseif tol <= 0 || tol >= 0.1
21         error('Tolleranza inconsistente.');
```

```

22     end
23     fa = feval(f, a);
24     fb = feval(f, b);
25     f1 = feval(f, x1);
26 end
27 h = (b - a) / 6;
28 x2 = (a + x1) / 2;
29 f2 = feval(f, x2);
30 x3 = (x1 + b) / 2;
31 f3 = feval(f, x3);
32 I2 = h * (fa + 4 * f1 + fb);
33 I4 = 0.5 * h * (fa + 4 * f2 + 2 * f1 + 4 * f3 + fb);
34 e = abs(I4 - I2) / 15;
35 if e > tol
36     I4 = adapsimp(f, a, x1, tol / 2, fa, f1, f2) + ...
37         adapsimp(f, x1, b, tol / 2, f1, fb, f3);
38 end
39 return

```

## 5.2 Esercizio 23

Sapendo che

$$I(f) = \int_0^{\text{atan}(30)} (1 + \tan^2(x)) dx = 30,$$

tabulare il numero di punti richiesti dalle formule adattative dei trapezi e di Simpson per approssimare  $I(f)$  con tolleranze

$$\text{tol} = 10^{-i}, i = 2, \dots, 8,$$

assieme ai relativi errori.

### Codice Matlab

```

1 f = @(x) 1 + tan(x)^2;
2 global pt;
3 global ps;
4 a = 0;
5 b = atan(30);
6 format long e;
7 for i = 2 : 8
8     pt = 2;
9     ps = 3;
10    tol = 10^-i;
```



```

11 I2 = adaptrap(f,a,b,tol);
12 I4 = adapsimp(f,a,b,tol);
13 pt
14 et = abs(I2 - 30)
15 ps
16 es = abs(I4 - 30)
17 end

```

#### Risultati

	<b>Trapezi</b>		<b>Simpson</b>	
Tolleranza	Punti	Errore	Punti	Errore
$10^{-2}$	375	4.764530169460102e-03	49	2.372538178065042e-03
$10^{-3}$	1181	5.737280523554489e-04	77	6.165282965362451e-04
$10^{-4}$	3687	5.263954536971482e-05	137	5.016670867874495e-05
$10^{-5}$	11883	5.454611955002520e-06	249	2.220652294937508e-06
$10^{-6}$	37273	5.635538471437940e-07	425	3.981788339046943e-07
$10^{-7}$	116747	5.255170520968022e-08	765	3.565320128018357e-08
$10^{-8}$	375793	5.507661882120374e-09	1365	3.587672381399898e-09

## 6 Capitolo 6

### 6.1 Esercizio 24

Scrivere una function che implementi efficientemente il metodo delle potenze.

---

Metodo delle potenze:

```

1 function [l1, x1, i] = mypower(A, x0, tol, imax)
2 %
3 % [l1, x1, i] = mypower(A[, x0[, tol[, imax]]) calcola l'autovalore dominante
4 %                                     semplice della matrice in input,
5 %                                     utilizzando il metodo delle potenze
6 %
7 [m, n] = size(A);
8 if m ~= n, error('Matrice non quadrata. '), end
9 if nargin <= 3
10     if nargin <= 2
11         if nargin <= 1
12             x0 = rand(n, 1);
13         end
14         tol = 1E-6;
15     elseif tol <= 0 || tol >= 0.1
16         error('Tolleranza inconsistente. ');
17     end
18     imax = ceil(-log10(tol));
19 end
20 x = x0;
21 l1 = 0;
22 for i = 1 : imax
23     x1 = x / norm(x);
24     x = A * x1;
25     l0 = l1;
26     l1 = x1' * x;
27     e = abs(l1 - l0);
28     if e <= tol * (1 + abs(l1)), break, end
29 end
30 if e > tol * (1 + abs(l1)), warning('Autovalore non trovato nella tolleranza specificata.
    '), end
31 return

```

## 6.2 Esercizio 25

Sia data la matrice di *Toeplitz* simmetrica

$$A_N = \begin{pmatrix} 4 & -1 & & -1 & & \\ -1 & \ddots & \ddots & & \ddots & \\ & \ddots & \ddots & \ddots & & -1 \\ -1 & & \ddots & \ddots & \ddots & \\ & \ddots & & \ddots & \ddots & -1 \\ & & -1 & -1 & 4 & \end{pmatrix} \in \mathbb{R}^{N \times N}, N \geq 10,$$

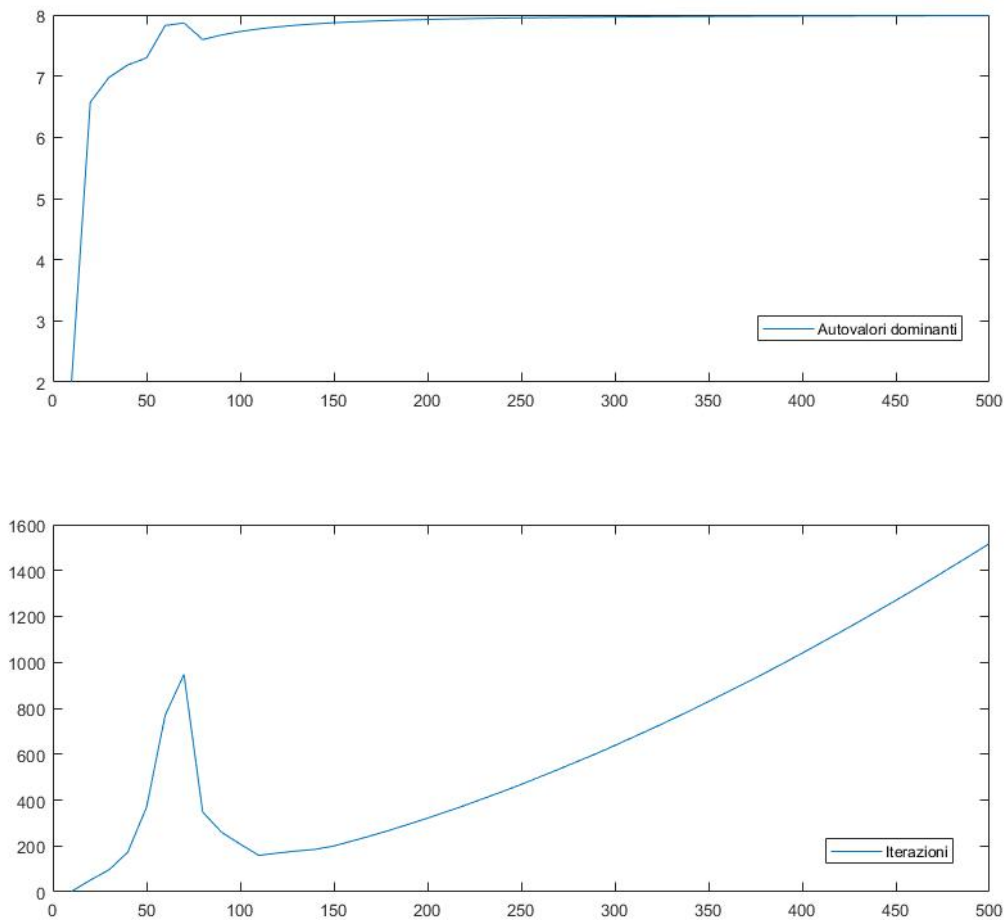
in cui le extra-diagonali più esterne sono le none. Partendo dal vettore  $\mathbf{u}_0 = (1, \dots, 1)^T \in \mathbb{R}^N$ , applicare il metodo delle potenze con tolleranza  $tol = 10^{-10}$  per  $N = 10 : 10 : 500$ , utilizzando la function del precedente esercizio. Graficare il valore dell'autovalore dominante, e del numero di iterazioni necessarie per soddisfare il criterio di arresto, rispetto ad  $N$ . Utilizzare la function **spdiags** di Matlab per creare la matrice e memorizzarla come matrice sparsa.

---

### Codice Matlab

```
1 x = linspace(10, 500, 50);
2 ax1 = subplot(2, 1, 1);
3 ax2 = subplot(2, 1, 2);
4 l = zeros(50, 1);
5 it = zeros(50, 1);
6 tol = 1E-10;
7 for n = 10 : 10 : 500
8     u = ones(n, 1);
9     S = spdiags(u * [-1 -1 4 -1 -1], [-9, -1 : 1, 9], n, n);
10    [l1, x1, i] = mypower(S, u, tol);
11    l(n / 10) = l1;
12    it(n / 10) = i;
13 end
14 plot(ax1, x, l);
15 plot(ax2, x, it);
16 legend(ax1, 'Autovalori dominanti');
17 legend(ax2, 'Iterazioni');
```

Le seguenti figure mostrano gli autovalori dominanti e le iterazioni necessarie al metodo delle potenze, al variare della dimensione della matrice  $A_N$  con  $N = 10 : 10 : 500$ :



Autovalori dominanti e iterazioni del metodo delle potenze con  $N = 10 : 10 : 500$

### 6.3 Esercizio 26

Scrivere una function che implementi efficientemente un metodo iterativo, per risolvere un sistema lineare, definito da un generico splitting della matrice dei coefficienti.

**Risoluzione iterativa del sistema lineare definito dallo splitting  $A = M - N$ :**

```

1 function [x, i] = itersolve(M, N, b, x0, tol, imax)
2 %
3 % [x, i] = itersolve(M, N, b [,x0 [, tol [, imax]]]) risolve il sistema lineare, con
4 % matrice dei coefficienti definita
5 % da un generico splitting  $M - N$ , in
6 % modo iterativo
7 %
8 n = length(b);
9 if rank(M) ~= n, error('Matrice M singolare.'), end
10 if nargin <= 5
11     if nargin <= 4

```

```

12         if nargin <= 3
13             x0 = rand(n, 1);
14         end
15         tol = 1E-6;
16     elseif tol <= 0 || tol >= 0.1
17         error('Tolleranza inconsistente.');
```

```

18     end
19     imax = n * max(1, ceil(-log10(tol))) * 100;
20 end
21 x = x0;
22 A = M - N;
23 for i = 1 : imax
24     r = A * x - b;
25     nr = norm(r, inf);
26     if nr <= tol, break, end
27     x = x - M\r;
28 end
29 if nr > tol, warning('Soluzione non trovata nella tolleranza specificata. '), end
30 return

```

**Risoluzione iterativa del sistema lineare con il metodo msolve:**

```

1 function [x, i] = splitting(b, matvec,msolve, x0, tol, imax)
2 %
3 % [x, i] = itersolve(b, matvec, msolve [,x0 [, tol [, imax]]]) risolve il sistema
4 %                                                                 lineare con il metodo
5 %                                                                 iterativo definito dalla
6 %                                                                 function msolve
7 %
8 n = length(b);
9 if nargin <= 5
10     if nargin <= 4
11         if nargin <= 3
12             x0 = zeros(n, 1);
13         end
14         tol = 1E-6;
15     elseif tol <= 0 || tol >= 0.1
16         error('Tolleranza inconsistente.');
```

```

17     end
18     imax = n * max(1, ceil(-log10(tol))) * 100;
19 end
20 x = x0;
21 tolb = tol * norm(b);
22 for i = 1 : imax
23     r = matvec(x) - b;
24     nr = norm(r);
25     if nr <= tolb, break, end
26     x = x - msolve(r);
27 end
28 if nr > tolb, warning('Soluzione non trovata nella tolleranza specificata. '), end
29 return

```

## 6.4 Esercizio 27

Scrivere le function ausiliarie, per la function del precedente esercizio, che implementano i metodi iterativi di Jacobi e Gauss-Seidel.

### Metodo di Jacobi:

```
1 function [x, i] = jacobi(A, b, x0, tol, imax)
2 %
3 % [x, i] = jacobi(A, b[, x0[, tol[, imax]]) risolve il sistema lineare
4 %                               Ax = b, con il metodo iterativo
5 %                               di Jacobi
6 %
7 [m, n] = size(A);
8 if m ~= n, error('Matrice non quadrata. '), end
9 if n ~= length(b), error('Vettore dei termini noti inconsistente. '), end
10 M = diag(diag(A));
11 N = M - A;
12 if nargin <= 2
13     [x, i] = itersolve(M, N, b);
14 elseif nargin <= 3
15     [x, i] = itersolve(M, N, b, x0);
16 elseif nargin <= 4
17     [x, i] = itersolve(M, N, b, x0, tol);
18 elseif nargin <= 5
19     [x, i] = itersolve(M, N, b, x0, tol, imax);
20 end
21 return
```

### Metodo di Gauss-Seidel:

```
1 function [x, i] = gs(A, b, x0, tol, imax)
2 %
3 % [x, i] = gs(A, b[, x0[, tol[, imax]]) risolve il sistema lineare
4 %                               Ax = b, con il metodo iterativo
5 %                               di Gauss-Seidel
6 %
7 [m, n] = size(A);
8 if m ~= n, error('Matrice non quadrata. '), end
9 if n ~= length(b), error('Vettore dei termini noti inconsistente. '), end
10 M = tril(A);
11 N = M - A;
12 if nargin <= 2
13     [x, i] = itersolve(M, N, b);
14 elseif nargin <= 3
15     [x, i] = itersolve(M, N, b, x0);
16 elseif nargin <= 4
17     [x, i] = itersolve(M, N, b, x0, tol);
18 elseif nargin <= 5
19     [x, i] = itersolve(M, N, b, x0, tol, imax);
20 end
21 return
```

## 6.5 Esercizio 28

Con riferimento alla matrice  $A_N$  definita in (1), risolvere il sistema lineare

$$A_N \mathbf{x} = \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} \in \mathbb{R}^N,$$

con i metodi di Jacobi e Gauss-Seidel, per  $N = 10 : 10 : 500$ , partendo dalla approssimazione nulla della soluzione, ed imponendo che la norma del residuo sia minore di  $10^{-8}$ . Utilizzare, a tal fine, la function dell'esercizio 26, scrivendo function ausiliarie *ad hoc* (vedi esercizio 27) che sfruttino convenientemente la struttura di sparsità (nota) della matrice  $A_N$ . Graficare il numero delle iterazioni richieste dai due metodi iterativi, rispetto ad  $N$ , per soddisfare il criterio di arresto prefissato.

#### Prodotto *ad hoc* matrice vettore:

```

1 function y = matvec(x)
2 %
3 % y = matvec(x) calcola il prodotto ad hoc
4 %           matrice vettore
5 %
6 y = 4 * x;
7 y(1 : end - 1) = y(1 : end - 1) - x(2 : end);
8 y(2 : end) = y(2 : end) - x(1 : end - 1);
9 y(1 : end - 8) = y(1 : end - 8) - x(9 : end);
10 y(9 : end) = y(9 : end) - x(1 : end - 8);
11 return

```

#### Metodo di Jacobi:

```

1 function u = jacobi(r)
2 %
3 % u = jacobi(r) risolve il sistema lineare ad hoc
4 %           con il metodo iterativo di Jacobi
5 %
6 n = length(r);
7 if n < 10, error('Dimensione minima non rispettata. '), end
8 u = r / 4;
9 return

```

#### Metodo di Gauss-Seidel:

```

1 function u = gs(r)
2 %
3 % u = gs(r) risolve il sistema lineare ad hoc
4 %           con il metodo iterativo di Gauss-Seidel
5 %
6 n = length(r);
7 if n < 10, error('Dimensione minima non rispettata. '), end
8 u = r;
9 u(1) = u(1) / 4;
10 for i = 2 : 8
11     u(i) = (u(i) + u(i - 1)) / 4;
12 end
13 for i = 1: n - 8
14     u(i + 8) = (u(i + 8) + u(i) + u(i + 7)) / 4;
15 end
16 return

```

#### Codice Matlab

```

1 x = linspace(10, 500, 50);
2 tol = 1E - 8;
3 ij = zeros(50, 1);
4 igs = zeros(50, 1);

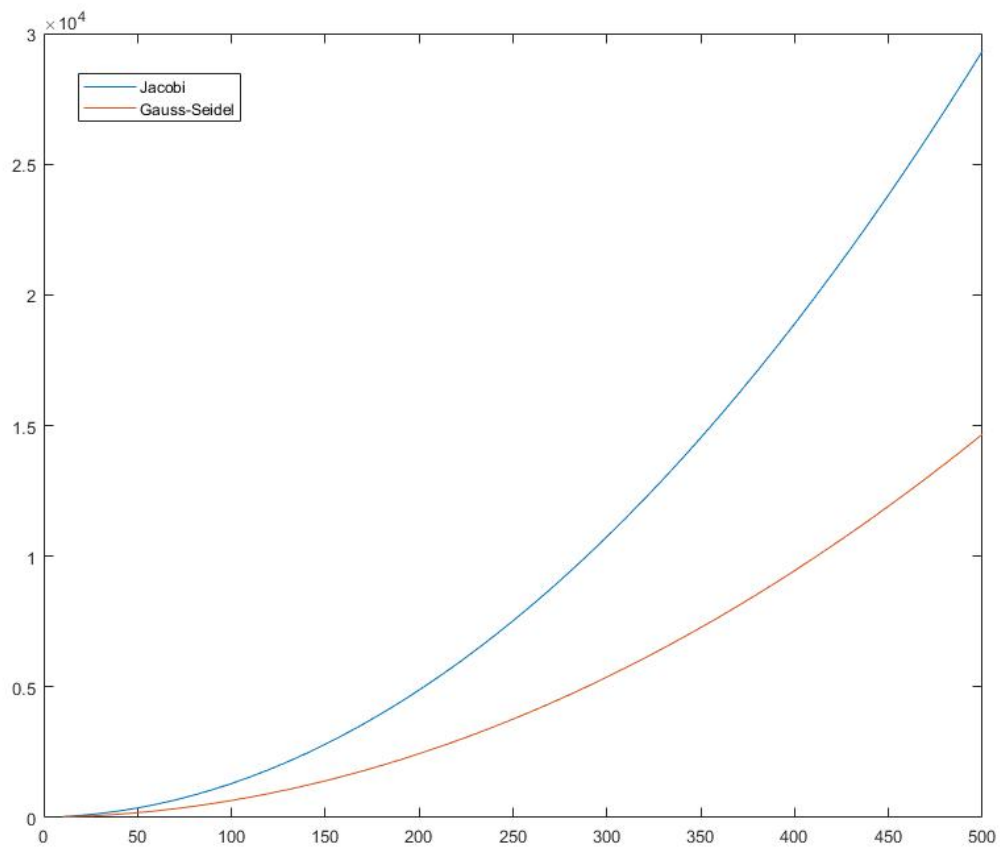
```

```

5  for n = 10 : 10 : 500
6      b = ones(n, 1);
7      x0 = zeros(n, 1);
8      [x, i] = splitting(b, @matvec, @jacobi, x0, tol);
9      ij(n / 10) = i;
10     [x, i] = splitting(b, @matvec, @gs, x0, tol);
11     igs(n / 10) = i;
12 end
13 plot(x, ij, x, igs);
14 legend('Jacobi', 'Gauss-Seidel');

```

La seguente figura mostra il confronto del numero di iterazioni necessarie ai metodi di *Jacobi* e *Gauss-Seidel*, al variare della dimensione della matrice  $A_N$  con  $N = 10 : 10 : 500$ :



Numero iterazioni dei metodi di *Jacobi* e *Gauss-Seidel* con  $N = 10 : 10 : 500$