Elaborato di

Calcolo Numerico

Anno Accademico 2017/2018

Daniel **Zanchi**, 5655418 — *daniel.zanchi@stud.unifi.it*

Capitolo 1

Esercizi A.A. 2017-18

Esercizio 1.1 *Sia $x = e \approx 2.7183 = \tilde{x}$. Si calcoli il corrispondente errore relativo ε_x e il numero di cifre significative k con cui $\tilde{x}$ approssima x. Si verifichi che*

$$|\varepsilon_x| \approx \frac{1}{2}10^{-k}.$$

Soluzione: Ricordiamo la definizione di *errore relativo*

$$\varepsilon_x = \frac{\tilde{x} - x}{x}$$

Il numero di Nepero e in MatLab rappresentato tramite la funzione *exp(1)* è:

$$2.718281828459046$$

Applicando la definizione abbiamo:

$$|\varepsilon_x| = \frac{|2.7183 - 2.718281828459046|}{|2.718281828459046|} = 6.684936331611679 * 10^{-6}$$

Adesso si verifica che:

- $\tilde{x} = 2.7183$ approssima e con $k = 5$ cifre significative
- $\frac{1}{2}10^{-k} = 0.5 \times 10^{-5} \approx 6.68 \times 10^{-6} = |\varepsilon_x|$

Esercizio 1.2 *Usando gli sviluppi di Taylor fino al secondo ordine con resto in forma di Lagrange, si verifichi che se $f \in C^3$, risulta*

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

dove

$$\phi_h(x) = \frac{f(x+h) - f(x-h)}{2h}$$

Soluzione: Sia $f \in C^3$ e sia $f_T(x)$ l'approssimazione al secondo ordine di f mediante il polinomio di Taylor con resto di Lagrange centrato nel punto x_0.
Ricordando che:

$$P_n(x) = \sum_{k=0}^{n} \frac{f^{(k)}(x_0)}{k!}(x - x_0)^k$$

$$R_{n,x_0}(x) = \frac{f^{(n+1)}(c)}{(n+1)!}(x-x_0)^{n+1}$$

Abbiamo:

$$f_T(x) = P_2(x) + R_{2,x_0}(x)$$

$$f_T(x) = f(x_0) + f'(x_0)(x-x_0) + \frac{f''(x_0)}{2}(x-x_0)^2 + \frac{f'''(c)}{6}(x-x_0)^3$$

Quindi, considerando il rapporto incrementale che definisce $f'(x)$:

$$f_T(x+h) = f(x) + f'(x)h + \frac{1}{2}f''h^2 + \frac{f'''(c)}{6}h^3$$

$$f_T(x-h) = f(x) - f'(x)h + \frac{1}{2}f''h^2 + \frac{f'''(c)}{6}h^3$$

Si noti che tutte derivate che compaiono in $f_T(x)$ esistono dato che $f \in C^3$.
Si procede ora a mostrare che $f'(x) = \frac{f(x+h)-f(x-h)}{2h} + O(h^2)$ sostituendo f con f_T nel rapporto incrementale.

$$f'(x) = \frac{f_T(x+h) - f_T(x-h)}{2h}$$

$$= \frac{f(x) + f'(x)h + \frac{1}{2}f''(x)h^2 + \frac{f'''(c)}{6}h^3 - f(x) + f'(x)h - \frac{1}{2}f''(x)h^2 + \frac{f'''(c)}{6}h^3}{2h}$$

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

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

$$= f'(x) + \frac{f'''(c)}{6}h^2$$

Esprimiamo il termine che rappresenta il resto di Lagrange $\frac{f'''(c)}{6}h^2$ tramite la notazione $O(h^2)$ ed otteniamo la tesi.

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

Esercizio 1.3 *Utilizzando Matlab, si costruisca una tabella dove, per* $h = 10^{-j}$, $j = 1, ..., 10$ *e per la funzione* $f(x) = x^4$ *si riporta il valore di* $\phi_h(x)$ *definito nell'esercizio 1 in* $x = 1$. *Commentare i risultati ottenuti.*

Soluzione:

```
function [] = es1_32(x, itmax)
    % -x: valore passato alla funzione
    % -itmax: massimo numero di iterazione
    format long e
    i = 1;
    while (i<=itmax)
        h = 10^-i;
        f = ((x+h)^4-(x-h)^4)/(2*h);
        str = sprintf('x%d =', -i);
        disp(str), disp(f)
        i = i+1;
    end
end
```

Funzione che produce i seguenti risultati:

h	$\phi_h(1)$
10^{-1}	4.040000000000002
10^{-2}	4.000400000000004
10^{-3}	4.000003999999723
10^{-4}	4.000000039999230
10^{-5}	4.000000000403681
10^{-6}	3.999999999948489
10^{-7}	4.000000000115023
10^{-8}	4.000000003445692
10^{-9}	4.000000108916879
10^{-10}	4.000000330961484

Si nota che all'aumentare di i, quindi al diminuire di h, ϕ_h diminuise che sta a significare un aumento di precisione del risultato approssimato.

Esercizio 1.4 *Si dia una maggiorazione del valore assoluto dell'errore relativo con cui* $x + y + z$ *viene approssimato dall'approssimazione prodotta dal calcolatore, ossia* $(x \oplus y) \oplus z$ *(supporre che non ci siano problemi di overflow o di underflow). Ricavare l'analoga maggiorazione anche per* $x \oplus (y \oplus z)$ *tenendo presente che* $x \oplus (y \oplus z) = (y \oplus z) \oplus x$.

Soluzione: Ricordiamo che l'apposimazione $(x \oplus y) \oplus z$ sarà equivalente al seguente errore:

$$\varepsilon_1 = \frac{(x\varepsilon_x + y\varepsilon_y) + z\varepsilon_z}{(x + y) + z}$$

Chiamando ε_{max} il massimo tra ε_x, ε_y e ε_z otteniamo:

$$\varepsilon_1 = \frac{|(x + y)| + |z|}{|(x + y) + z|}\varepsilon_{max}$$

Per lo stesso motivo $x \oplus (y \oplus z)$ sarà :

$$\varepsilon_2 = \frac{|x| + |(y+z)|}{|x + (y+z)|}\varepsilon_{max}$$

Esercizio 1.5 *Eseguire le seguenti operazioni in Matlab:*

$x = 0$; *count* $= 0$;

while $x \sim 1 = 1$, $x = x +$ *delta*, *count* $=$ *count* $+ 1$, *end*

dapprima ponendo delta = 1/16 e poi ponendo delta = 1/20. Commentare i risultati ottenuti e in particolare il non funzionamento nel secondo caso.

Soluzione: L'algoritmo viene eseguito correttamente se poniamo *delta* $= 1/16$.
Invece ponendo *delta* $= 1/20$ il codice va in loop:
Questo è dovuto perchè la rappresentazione di 1/20 (0.05) in binario equivale a $0.00\overline{0011}$. Essendo *delta* periodico, deve essere approssimato. Questo errore di approssimazione comporta a diventare sempre più grosso dopo ogni iterazione. L'errore di approssimazione fa ottenere un valore diverso da 1. Ecco perchè non sarà mai uguale a 1.

Esercizio 1.6 *Verificare che entrambe le seguenti successioni convergono a* $\sqrt{3}$*, (riportare le successive approssimazioni in una tabella a due colonne, una per ciascuna successione),*

$$x_{k+1} = (x_k + \tfrac{3}{x_k})/2, \qquad x_0 = 3;$$
$$x_{k+1} = (3 + x_{k-1}x_k)/(x_{k-1} + x_k), \qquad x_0 = 3;\ x_1 = 2.$$

Per ciascuna delle due successioni, dire quindi dopo quante iterazioni si ottiene un'approssimazione con un errore assoluto minore o uguale a 10^{-12} *in valore assoluto.*

Soluzione: Entrambe le successioni convergono a $\sqrt{3}$. Notiamo che la prima successione converge più velocemente; infatti dopo cinque iterazioni abbiamo come risultato 1.7320508075689:

Prima successione:

k	$x(k)$
0	3.0000000000000
1	2.0000000000000
2	1.7500000000000
3	1.7321428571429
4	1.7320508100147
5	1.7320508075689

Seconda successione:

k	$x(k)$
0	3.0000000000000
1	2.0000000000000
2	1.8000000000000
3	1.7368421052632
4	1.7321428571429
5	1.7320509347060
6	1.7320508075723
7	1.7320508075689

Per rispondere alla seconda domanda ricordiamo la definizione di valore assoluto:

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

Dato che si tratta di una successione, parliamo di errore assoluto di convergenza commesso ad ogni passo dell'iterazione, con x_k risultato intermedio ed x valore da approssimare ($\sqrt{3}$):

$$\Delta x_k = x_k - x$$

Per svolgere i conti è stato utilizzato il seguente script di Matlab dove è stato calcolato il valore assoluto ad ogni iterazione:

```
clc;
format long
f1 = @(x) (x + 3/x)/2; % Prima successione
f2 = @(x0, x1) (3 + x0*x1)/(x0 + x1); % Seconda successione
x = sqrt(3); % Valore da convergere

xk = 3; %Innesco per la successione f1

k = 1;
e = 10^(-12);
diff = abs(xk - sqrt(3));

disp('Prima successione: ');
fprintf('k = 0\t\tx(0) = %.13f\t err = %.15f', xk, diff);

while ((e <= diff) && xk ~= sqrt(3))
   xk = f1(xk);
   diff = abs(xk - sqrt(3));
   fprintf('\nk = %d\t\tx(%d) = %.13f\t err = %.15f', k, k, xk, (diff));
   k = k + 1;
end

fprintf(successione:);
x0 = 3; %innesco per succeccione f2
x1 = 2; %innesco per successione f2
xk = 2; %innesco che cambiera nel ciclo
k = 2;
diff = x0 - sqrt(3);
fprintf('k = 0\t\tx(0) = %.13f \t err = %.15f', x0, diff);
diff = abs(x1 - sqrt(3));
fprintf('\nk = 1\t\tx(1) = %.13f \t err = %.15f', x1, diff);
while ((e <= diff) && (x0 ~= sqrt(3)))
   xk = f2(x0, x1);
   diff = abs(xk - sqrt(3));
   fprintf('\nk = %d\t\tx(%d) = %.13f\t err = %.15f', k, k, xk, (diff));
   x0 = x1;
   x1 = xk;
   k = k + 1;
end
```

```
Prima successione:
k = 0           x(0) = 3.0000000000000   err = 1.267949192431123
k = 1           x(1) = 2.0000000000000   err = 0.267949192431123
k = 2           x(2) = 1.7500000000000   err = 0.017949192431123
k = 3           x(3) = 1.7321428571429   err = 0.000092049573980
k = 4           x(4) = 1.7320508100147   err = 0.000000002445850
k = 5           x(5) = 1.7320508075689   err = 0.000000000000000

Seconda successione:
k = 0           x(0) = 3.0000000000000   err = 1.267949192431123
k = 1           x(1) = 2.0000000000000   err = 0.267949192431123
k = 2           x(2) = 1.8000000000000   err = 0.067949192431123
k = 3           x(3) = 1.7368421052632   err = 0.004791297694281
k = 4           x(4) = 1.7321428571429   err = 0.000092049573980
k = 5           x(5) = 1.7320509347060   err = 0.000000127137164
k = 6           x(6) = 1.7320508075723   err = 0.000000000003379
k = 7           x(7) = 1.7320508075689   err = 0.000000000000000>>
```

Capitolo 2

Esercizi A.A. 2017-18

Esercizio 2.1 *Determinare analiticamente gli zeri del polinomio*

$$P(x) = x^3 - 4x^2 + 5x - 2$$

e la loro molteplicità. Dire perché il metodo di bisezione è utilizzabile per approssimarne uno a partire dall'intervallo di confidenza $[a, b] = [0, 3]$. *A quale zero di P potrà tendere la successione generata dal metodo di bisezione a partire da tale intervallo? Costruire una tabella in cui si riportano il numero di iterazioni e di valutazioni di P richieste per valori decrescenti della tolleranza* tolx.

Soluzione: Prendiamo in considerazione il nostro polinomio e vediamo subito che un suo zero è 1, infatti:
con $\hat{x} = 1$:

$$P(1) = 1^3 - 4 * 1^2 + 5 * 1 - 2 =$$
$$= 1 - 4 + 5 - 2 = 0$$

A questo punto controlliamo la sua molteplicità:

$$P'(x) = 3x^2 - 8x + 5$$
$$P'(1) = 3 - 8 + 5 = 0$$

$$P''(x) = 6x - 8$$
$$P''(1) = 6 - 8 = -2$$

La molteplicità è 2 poiché la derivata seconda è la prima derivata in ordine che non viene annullata.

Un'altro suo zero è 2:
con $\hat{x} = 2$:

$$P(2) = 8 - 16 + 10 - 2 = 0$$

$$P'(2) = 12 - 16 + 5 = 1$$

In questo caso la molteplicità è 1.

Il metodo di bisezione è utilizzabile nell'intervallo di confidenza $[0, 3]$ in quanto $P(a)P(b) < 0$, precisando che in tale intervallo P tenderà a 2:

$$P(0) = 0 - 0 + 0 - 2 = -2$$
$$P(3) = 27 - 36 + 15 - 2 = 4$$
$$-2 * 4 < 0$$

Di seguito i risultati dell'esecuzione del metodo di bisezione utilizzando come intervallo di confidenza $[0, 3]$:

$P(x) = x^3 - 4x^2 + 5x - 2, \qquad I = [0, 3]$		
$\tilde{x} \approx 2.0$		
tol_x	Approssimazione	Iterazioni
10^{-1}	$\tilde{x} = 2.062500000000000$	$i = 3$
10^{-2}	$\tilde{x} = 1.992187500000000$	$i = 6$
10^{-3}	$\tilde{x} = 2.000976562500000$	$i = 9$
10^{-4}	$\tilde{x} = 2.000061035156250$	$i = 13$
10^{-5}	$\tilde{x} = 1.999992370605469$	$i = 16$
10^{-6}	$\tilde{x} = 2.000000953674316$	$i = 19$
10^{-7}	$\tilde{x} = 2.000000059604645$	$i = 23$
10^{-8}	$\tilde{x} = 1.999999992549419$	$i = 26$
10^{-9}	$\tilde{x} = 2.000000000931323$	$i = 29$
10^{-10}	$\tilde{x} = 2.000000000058208$	$i = 33$
10^{-11}	$\tilde{x} = 1.999999999992724$	$i = 36$
10^{-12}	$\tilde{x} = 2.000000000000909$	$i = 39$
10^{-13}	$\tilde{x} = 2.000000000000057$	$i = 43$
10^{-14}	$\tilde{x} = 1.999999999999993$	$i = 46$
10^{-15}	$\tilde{x} = 2.000000000000001$	$i = 49$

Notiamo come dopo 49 iterazioni otteniamo il risultato con tolleranza pari a 10^{-15}.

Esercizio 2.2 *Completare la tabella precedente riportando anche il numero di iterazioni e di valutazioni di P richieste dal metodo di Newton, dal metodo delle corde e dal metodo delle secanti (con secondo termine della successione ottenuto con Newton) a partire dal punto* $x_0 = 3$. *Commentare i risultati riportati in tabella. E' possibile utilizzare* $x_0 = 5/3$ *come punto di innesco?*
Soluzione:

$P(x) = x^3 - 4x^2 + 5x - 2, \qquad x_0 = 3$			
$\tilde{x} \approx 2.0$			
tol_x	Newton	Corde	Secanti
10^{-1}	$\tilde{x} = 2.00435, i = 4$	$x = 2.35938, i = 3$	$x = 2.05016, i = 4$
10^{-2}	$\tilde{x} = 2.00004, i = 5$	$x = 2.06432, i = 13$	$x = 2.00099, i = 6$
10^{-3}	$\tilde{x} = 2.00000, i = 6$	$x = 2.00767, i = 29$	$x = 2.00002, i = 7$
10^{-4}	$\tilde{x} = 2.00000, i = 6$	$x = 2.00078, i = 47$	$x = 2.00000, i = 8$
10^{-5}	$\tilde{x} = 2.00000, i = 7$	$x = 2.00007, i = 66$	$x = 2.00000, i = 9$
10^{-6}	$\tilde{x} = 2.00000, i = 7$	$x = 2.00001, i = 84$	$x = 2.00000, i = 9$
10^{-7}	$\tilde{x} = 2.00000, i = 7$	$x = 2.00000, i = 102$	$x = 2.00000, i = 9$
10^{-8}	$\tilde{x} = 2.00000, i = 7$	$x = 2.00000, i = 120$	$x = 2.00000, i = 10$
10^{-9}	$\tilde{x} = 2.00000, i = 8$	$x = 2.00000, i = 139$	$x = 2.00000, i = 10$
10^{-10}	$\tilde{x} = 2.00000, i = 8$	$x = 2.00000, i = 157$	$x = 2.00000, i = 10$
10^{-11}	$\tilde{x} = 2.00000, i = 8$	$x = 2.00000, i = 175$	$x = 2.00000, i = 10$
10^{-12}	$\tilde{x} = 2.00000, i = 8$	$x = 2.00000, i = 193$	$x = 2.00000, i = 11$
10^{-13}	$\tilde{x} = 2.00000, i = 8$	$x = 2.00000, i = 211$	$x = 2.00000, i = 11$
10^{-14}	$\tilde{x} = 2.00000, i = 8$	$x = 2.00000, i = 230$	$x = 2.00000, i = 11$
10^{-15}	$\tilde{x} = 2.00000, i = 8$	$x = 2.00000, i = 247$	$x = 2.00000, i = 11$

Possiamo notare come il metodo più efficiente è quello di Newton, mentre il metodo meno efficiente è quello delle corde.
No non è possibile utilizzare $x_0 = 5/3$ come punto di innesco in quanto $5/3$ è uno zero della derivata prima.

Esercizio 2.3 *Costruire una seconda tabella analoga alla precedente relativa ai metodi di Newton, di Newton modificato e di accelerazione di Aitken applicati alla funzione polinomiale P a partire dal punto di innesco $x_0 = 0$. Commentare i risultati riportati in tabella.*

Soluzione: Avendo valutato precedentemente che la molteplicità di P è 2 applichiamo il metodo di Newton modificaton con coefficente del termine di correzione $m = 2$:

$P(x) = x^3 - 4x^2 + 5x - 2, \qquad x_0 = 0$			
$\tilde{x} \approx 1.0$			
tol_x	Newton	Newton modificato	Aitken
10^{-1}	$\tilde{x} = 0.89599, i = 4$	$x = 0.99607, i = 3$	$x = 1.00056, i = 3$
10^{-2}	$\tilde{x} = 0.99289, i = 8$	$x = 0.99999, i = 4$	$x = 1.00000, i = 4$
10^{-3}	$\tilde{x} = 0.99911, i = 11$	$x = 1.00000, i = 5$	$x = 1.00000, i = 4$
10^{-4}	$\tilde{x} = 0.99994, i = 15$	$x = 1.00000, i = 5$	$x = 1.00000, i = 5$
10^{-5}	$\tilde{x} = 0.99999, i = 18$	$x = 1.00000, i = 5$	$x = 1.00000, i = 5$
10^{-6}	$\tilde{x} = 1.00000, i = 21$	$x = 1.00000, i = 6$	$x = 1.00000, i = 5$
10^{-7}	$\tilde{x} = 1.00000, i = 25$	$x = 1.00000, i = 6$	$x = 1.00000, i = 5$
10^{-8}	$\tilde{x} = 1.00000, i = 29$	$x = 1.00000, i = 6$	$x = 1.00000, i = 6$
10^{-9}	$\tilde{x} = 1.00000, i = 29$	$x = 1.00000, i = 6$	$x = 1.00000, i = 6$
10^{-10}	$\tilde{x} = 1.00000, i = 29$	$x = 1.00000, i = 6$	$x = 1.00000, i = 6$
10^{-11}	$\tilde{x} = 1.00000, i = 29$	$x = 1.00000, i = 6$	$x = 1.00000, i = 6$
10^{-12}	$\tilde{x} = 1.00000, i = 29$	$x = 1.00000, i = 6$	$x = 1.00000, i = 6$
10^{-13}	$\tilde{x} = 1.00000, i = 29$	$x = 1.00000, i = 6$	$x = 1.00000, i = 6$
10^{-14}	$\tilde{x} = 1.00000, i = 29$	$x = 1.00000, i = 6$	$x = 1.00000, i = 6$
10^{-15}	$\tilde{x} = 1.00000, i = 29$	$x = 1.00000, i = 6$	

Come era prevedibile notiamo che il metodo di Newton converge linearmente. Il metodo di Newton modificato e quello di Aitken performano similmente e notiamo che il costo di discosta di poco.

Esercizio 2.4 *Definire una procedura iterativa basata sul metodo di Newton per approssimare $\sqrt{\alpha}$, per un assegnato $\alpha > 0$. Costruire una tabella dove si riportano le successive approssimazioni ottenute e i corrispondenti errori assoluti (usare l'approssimazione di Matlab di $\sqrt{\alpha}$ per il calcolo dell'errore) nel caso in cui $\alpha = 5$ partendo da $x_0 = 5$.*

Soluzione: Il seguente metodo produrrà la tabella richiesta:

```
alpha = 5;
x0 = 5;

Tabella = cell2table(cell(0,3));
Tabella.Properties.VariableNames = {'i' 'SQRT_a' 'err'};
[Tabella, sqrt_a] = SQRT_Newton(alpha, x0, 100, 10^-15, Tabella);

function [T, sqrt_a] = SQRT_Newton(alpha, x0, itmax, tolx, T);
    sqrt_a = (x0 + alpha/x0) / 2;
    i = 1;
    row = {i, sqrt_a, abs( sqrt(alpha) - sqrt_a )};
    T = [T; row];

    while (i < itmax) && (abs(sqrt_a-x0) > tolx)
        x0 = sqrt_a;
        i = i+1;
        sqrt_a = (x0 + alpha/x0) / 2;
        row = {i, sqrt_a, abs( sqrt(alpha) - sqrt_a )};
        T = [T; row];
    end
end
```

1 i	2 SQRT_a	3 err
1	3	0.7639
2	2.3333	0.0973
3	2.2381	0.0020
4	2.2361	9.1814e-07
5	2.2361	1.8829e-13
6	2.2361	0

Esercizio 2.5 *Definire una procedura iterativa basata sul metodo delle secanti sempre per approssimare $\sqrt{\alpha}$, per un assegnato $\alpha > 0$. Completare la tabella precedente aggiungendovi i risultati ottenuti con tale precedura partendo da $x_0 = 5$ e $x_1 = 3$. Commentare i risultati riportati in tabella.*

Soluzione:

```
x_0 = 5;
alpha = 5;

Tabella = cell2table(cell(0,3));
Tabella.Properties.VariableNames = {'i' 'sqrt_a' 'err'};

[Tabella, res] = SQRT_secanti(alpha, x_0, 200, 10^(-15), Tabella);

function [T, sqrt_alpha] = SQRT_secanti(alpha, x0, itmax, tolx, T)
x1 = (x0 + alpha/x0)/2;
x = ( (x1^2-alpha) * x0-(x0^2-alpha)*x1 ) / ((x1^2-alpha)-(x0^2 - alpha));
i = 1;
row = {i, x, abs( sqrt(alpha) - x )}; T = [T; row];
while(i < itmax) && (abs(x-x0)>tolx) x0=x1;
    x1=x;
    i = i+1;
    x = ( (x1^2 - alpha) * x0 - (x0^2 - alpha)*x1 ) / ((x1^2 - alpha) - (
       x0^2 - alpha));
    row = {i, x, abs( sqrt(alpha) - x )}; T = [T; row];
end
sqrt_alpha = x;
end
```

1 i	2 sqrt_a	3 err
1	2.5000	0.2639
2	2.2727	0.0367
3	2.2381	0.0020
4	2.2361	1.6475e-05
5	2.2361	7.4651e-09
6	2.2361	2.7089e-14
7	2.2361	4.4409e-16
8	2.2361	4.4409e-16
9	2.2361	0

Dal risultato ottenuto possiamo notare come a differenza del metodo di newton il metodo delle secanti converge $\sqrt{\alpha}$ meglio nelle prime iterazioni, ma il metodo di Newton converge poi con più precisione convergendo più velocemente verso il risultato esatto.

Capitolo 3

Esercizi A.A. 2017-18

Esercizio 3.1 *Scrivere una function Matlab per la risoluzione di un sistema lineare con matrice dei coefficienti triangolare inferiore a diagonale unitaria. Inserire un esempio di utilizzo.*
Soluzione:

```
A = [1 0 0; 3 1 0; 4 1 2];
b = [1 2 3];
x = sistema_triang_inf(A, b);

function [b] = sistema_triang_inf(A, b)
    for j = 1 : length(A)
        b(j) = b(j)/A(j,j);
        for i = j+1 : length(A)
            b(i) = b(i)-A(i,j)*b(j);
        end
    end
end
```

Il codice risolve sistemi lineari con matrice dei coefficenti triangolare inferiore a diagonale unitaria dove in input viene presa la matrice dei coefficienti A e il vettore dei termini noti b restituendo vettore con le soluzioni.
Un esempio è il seguente:

$$\begin{bmatrix} 1 & 0 & 0 \\ 3 & 1 & 0 \\ 4 & 1 & 2 \end{bmatrix} \bar{x} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

Il vettore delle soluzioni calcolato dalla funzione è: $\begin{bmatrix} 1 & -1 & 0 \end{bmatrix}$.

Esercizio 3.2 *Utilizzare l'Algoritmo 3.6 del libro per stabilire se le seguenti matrici sono sdp o no,*

$$A_1 = \begin{pmatrix} 1 & -1 & 2 & 2 \\ -1 & 5 & -14 & 2 \\ 2 & -14 & 42 & 2 \\ 2 & 2 & 2 & 65 \end{pmatrix}, A_2 = \begin{pmatrix} 1 & -1 & 2 & 2 \\ -1 & 6 & -17 & 3 \\ 2 & -17 & 48 & -16 \\ 2 & 3 & -16 & 4 \end{pmatrix}$$

Soluzione: Il seguente codice implementa l'algoritmo richiesto e si evince che la matrice A_1 è simmetrica e difinita positiva mentre la matrice A_2 non lo è:

```
A1 = [1 -1 2 2; -1  5  -14 2; 2 -14 42 2; 2 2 2 65];
A2 = [1 -1 2 2; -1 6 -17 3; 2 -17 48 -16; 2 3 -16 4];

algoritmo_A1 = algoritmo36(A1);
algoritmo_A2 = algoritmo36(A2);

function [A] = algoritmo36(A)
    [m,n]=size(A);
    if A(1,1)<=0
       error('matrice non 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('matrice non 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
```

Esercizio 3.3 *Scrivere una function Matlab che, avendo in ingresso un vettore b contenente i termini noti del sistema lineare $Ax = b$ con A sdp e l'output dell'Algoritmo 3.6 del libro (matrice A riscritta nella porzione triangolare inferiore con i fattori L e D della fattorizzazione LDL^T di A), ne calcoli efficientemente la soluzione.*

Soluzione:

```
function [b] = sistema_lineare_LDLt(A, b)
% [b] = sistema_lineare_LDLt(A, b)
% Calcola la soluzione di Ax=b con
% A: matrice LDLt (da fattorizzazione LDLt)
% b: vettore colonna
% [b]: soluzione del sistema
    b = sistema_triang_inf(tril(A,-1)+eye(length(A)), b);
    b = diagonale(diag(A), b);
    b = sistema_triang_sup((tril(A,-1)+eye(length(A)))',b);
end

function [d] = diagonale(d, b)
% [d] = diagonale(d, b)
% Calcolare la soluzione di Ax=b con
% d: matrice diagonale
% b: vettore colonna
% [d]: soluzione del sistema
    n = size(d);
    for i = 1:n
        d(i) = b(i)/d(i);
```

```
    end
end

function [b] = sistema_triang_sup(A, b)
% [b] = sistema_tirnag_sup(A, b)
% Calcola la soluzione di Ax=b con
% A: matrice triangolare superiore
% b: vettore colonna
% [b]: soluzione del sistema
    for j = length(A) : -1 : 1
        b(j)=b(j)/A(j,j);
        for i = 1 : j-1
            b(i) = b(i)-A(i,j)*b(j);
        end
    end
end
```

La function `sistema_triang_inf` è stata implementata nell'esercizio 1 di questo capitolo.

Esercizio 3.4 *Scrivere una function Matlab che, avendo in ingresso un vettore b contenente i termini noti del sistema lineare $Ax = b$ e l'output dell'Algoritmo 3.7 del libro (matrice A riscritta con la fattorizzazione LU con pivoting parziale e il vettore p delle permutazioni), ne calcoli efficientemente la soluzione.*
Soluzione: Il seguente codice calcola quanto richiesto, le function `sistema_triang_inf` e `sistema_triang_sup` sono quelle degli Esercizi 1 e 3.

```
function [A, p] = fatt_LUpivot(A)
    % [A, p] = fattorizzaLUpivot(A)
    % Calcola la fattorizzazione LU della matrice A.
    % A: matrice da fattorizzare.
    % Output:
    % A: la matrice fattorizzata LU;
    % p: vettore di permutazione
    [m,n]=size(A);
    if m~=n
        error('Matrice non quadrata');
    end
    p=(1:n);
    for i=1:n-1
        [mi, ki] = max(abs(A(i:n, i)));
        if mi==0
            error('Matrice 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

function [b]= sistema_LUpivot(A,b,p)
    % [b]= sistema_LUpivot(A,b,p)
    % Calcola la soluzione di Ax=b con A matrice LU con pivoting parziale
    % A: Matrice matrice LU con pivoting (generata da fatt_LUpivot)
    % b: vettore colonna
    % Output:
    % b: soluzione del sistema
    P = zeros(length(A));
    for i = 1:length(A)
        P(i, p(i)) = 1;
    end
    b = sistema_triang_inf(tril(A,-1)+eye(length(A)), P*b);
    b = sistema_triang_sup(triu(A), b);
end
```

Esercizio 3.5 *Inserire alcuni esempi di utilizzo delle due function implementate per i punti 3 e 4, scegliendo per ciascuno di essi un vettore $\hat{x}$ e ponendo $b = A\hat{x}$. Riportare $\hat{x}$ e la soluzione x da essi prodotta. Costruire anche una tabella in cui, per ogni esempio considerato, si riportano il numero di condizionamento di A in norma 2 (usare **cond** di Matlab) e le quantità $||r||/||b||$ e $||x - \hat{x}||/||\hat{x}||$.*

Soluzione: Per dimostrare le function dell'esercizio 3 (risoluzione di sistemi LDL^T) e dell'esercizio 4 (scomposizione LU con pivoting parziale) verrà utilizzata la seguente matrice:

$$A = \begin{pmatrix} 1 & -1 & 2 & 2 \\ -1 & 5 & -14 & 2 \\ 2 & -14 & 42 & 2 \\ 2 & 2 & 2 & 65 \end{pmatrix}$$

I vettori scelti sono, rispettivamente per svolgere gli esercizi 3 e 4:

$$\hat{x}_3 = \begin{bmatrix} 5.1211 \\ 3.4433 \\ 0.1257 \\ 2.1579 \end{bmatrix}, \hat{x}_4 = \begin{bmatrix} 1.3345 \\ 2.3232 \\ 3.1175 \\ 1.6658 \end{bmatrix}$$

Ponendo $b = A\hat{x}$ risulterà rispettivamente:

$$b_3 = \begin{bmatrix} 6.2450 \\ 14.6514 \\ -28.3688 \\ 157.6437 \end{bmatrix}, b_4 = \begin{bmatrix} 8.5779 \\ -30.0319 \\ 104.4108 \\ 121.8274 \end{bmatrix}$$

Eseguendo le function create per la risoluzione del sistema viene generata la seguente soluzione:

$$x_3 = \begin{bmatrix} 5.12110000000000 \\ 3.44330000000000 \\ 0.125699999999998 \\ 2.15790000000000 \end{bmatrix}, x_4 = \begin{bmatrix} 1.33450000000002 \\ 2.32320000000003 \\ 3.11750000000001 \\ 1.66580000000000 \end{bmatrix}$$

La tabella seguente mostra il condizionamento della matrice restituita dagli algoritmi di fattorizzazione ed i vari confronti di errori relativi sui dati di ingresso ($||r||/||b||$) e sul risultato ($||x - \hat{x}||/||\hat{x}||$):

Fattorizzazione	$\hat{x}$	$cond(A, 2)$	$\|\|r\|\|/\|\|b\|\|$	$\|\|x - \hat{x}\|\|/\|\|\hat{x}\|\|$
LDL^T	$\hat{x}_3$	3.6158×10^3	4.4142×10^{-17}	8.3347×10^{-16}
LU pivot	$\hat{x}_4$	319.1025	9.0270×10^{-17}	8.4115×10^{-15}

Il codice Matlab usato per realizzare i precedenti esempi è il seguente:

```
A = [1 −1 2 2; −1  5  −14 2; 2 −14 42 2; 2 2 2 65];

A3 = A;
A4 = A;

x3 = [5.1211 3.4433 0.1257 2.1579]';
x4 = [1.3345 2.3232 3.1175 1.6658]';

b3 = A3 * x3;
b4 = A4 * x4;

% fattorizzazione LDL^T
x3_soluzione = sistema_lineare_LDLt(algoritmo36(A3), b3);
cond_A3_2 = cond(A3, 2);
r3 = A*x3_soluzione − b3;
r_b_3 = norm(r3) / norm(b3);
err_3 = norm(x3_soluzione − x3)/norm(x3_soluzione);

% fattorizzazione LU con pivoting parziale
[A4, p4] = fatt_LUpivot(A4);
x4_soluzione = sistema_LUpivot(A4, b4, p4);
cond_A4_2 = cond(A4, 2);
r4 = A*x4_soluzione − b4;
r_b_4 = norm(r4) / norm(b4);
err_4 = norm(x4_soluzione − x4)/norm(x4_soluzione);

function [b] = sistema_lineare_LDLt(A, b)
% [b] = sistema_lineare_LDLt(A, b)
% Calcola la soluzione di Ax=b con
% A: matrice LDLt (da fattorizzazione LDLt)
% b: vettore colonna
% [b]: soluzione del sistema
    b = sistema_triang_inf(tril(A,−1)+eye(length(A)), b);
    b = diagonale(diag(A), b);
    b = sistema_triang_sup((tril(A,−1)+eye(length(A)))',b);
```

```
36 end
37
38 function [A, p] = fatt_LUpivot(A)
39     % [A, p] = fattorizzaLUpivot(A)
40     % Calcola la fattorizzazione LU della matrice A.
41     % A: matrice da fattorizzare.
42     % Output:
43     % A: la matrice fattorizzata LU;
44     % p: vettore di permutazione
45     [m,n]=size(A);
46     if m~=n
47         error('Matrice non quadrata');
48     end
49     p=(1:n);
50     for i=1:n-1
51         [mi, ki] = max(abs(A(i:n, i)));
52         if mi==0
53             error('Matrice singolare');
54         end
55         ki = ki+i-1;
56         if ki>i
57             A([i ki], :) = A([ki i], :);
58             p([i ki]) = p([ki i]);
59         end
60         A(i+1:n, i) = A(i+1:n, i)/A(i, i);
61         A(i+1:n, i+1:n) = A(i+1:n, i+1:n) -A(i+1:n, i)*A(i, i+1:n);
62     end
63 end
64
65 function [A] = algoritmo36(A)
66     [m,n]=size(A);
67     if A(1,1)<=0
68        error('matrice non sdp');
69     end
70     A(2:n,1) = A(2:n,1)/A(1,1);
71     for j = 2:n
72         v = ( A(j,1:(j-1))') .* diag(A(1:(j-1),1:(j-1)));
73         A(j,j) = A(j,j)-A(j,1:(j-1))*v;
74         if A(j,j)<=0
75             error('matrice non sdp');
76         end
77         A((j+1):n,j)=(A((j+1):n,j)-A((j+1):n,1:(j-1))*v)/A(j,j);
78     end
79 end
80
81 function [b]= sistema_LUpivot(A,b,p)
82     % [b]= sistema_LUpivot(A,b,p)
83     % Calcola la soluzione di Ax=b con A matrice LU con pivoting parziale
```

```
84      % A: Matrice matrice LU con pivoting (generata da fatt_LUpivot)
85      % b: vettore colonna
86      % Output:
87      % b: soluzione del sistema
88      P = zeros(length(A));
89      for i = 1:length(A)
90          disp(i);
91          disp(p(i));
92          P(i, p(i)) = 1;
93      end
94      b = sistema_triang_inf(tril(A,-1)+eye(length(A)), P*b);
95      b = sistema_triang_sup(triu(A), b);
96  end
97
98  function [d] = diagonale(d, b)
99  % [d] = diagonale(d, b)
100 % Calcolare la soluzione di Ax=b con
101 % d: matrice diagonale
102 % b: vettore colonna
103 % [d]: soluzione del sistema
104     n = size(d);
105     for i = 1:n
106         d(i) = b(i)/d(i);
107     end
108 end
109
110 function [b] = sistema_triang_sup(A, b)
111 % [b] = sistema_tirnag_sup(A, b)
112 % Calcola la soluzione di Ax=b con
113 % A: matrice triangolare superiore
114 % b: vettore colonna
115 % [b]: soluzione del sistema
116     for j = length(A) : -1 : 1
117         b(j)=b(j)/A(j,j);
118         for i = 1 : j-1
119             b(i) = b(i)-A(i,j)*b(j);
120         end
121     end
122 end
123
124 function [b] = sistema_triang_inf(A, b)
125     for j = 1 : length(A)
126         b(j) = b(j)/A(j,j);
127         for i = j+1 : length(A)
128             b(i) = b(i)-A(i,j)*b(j);
129         end
130     end
131 end
```

Esercizio 3.6 *Sia* $A = \begin{pmatrix} \epsilon & 1 \\ 1 & 1 \end{pmatrix}$ *con* $\epsilon = 10^{-13}$. *Definire* L *triangolare inferiore a diagonale unitaria e* U *triangolare superiore in modo che il prodotto* LU *sia la fattorizzazione* LU *di* A *e, posto* $b = Ae$ *con* $e = (1, 1)^T$, *confrontare l'accuratezza della soluzione che si ottiene usando il comando* $U\backslash(L\backslash b)$ *(Gauss senza pivoting) e il comando* $A\backslash b$ *(Gauss con pivoting).*

Soluzione: Data la matrice A possiamo scrivere:

$$A = L \times U = \begin{bmatrix} 1 & 0 \\ 10^{13} & 1 \end{bmatrix} \times \begin{bmatrix} 10^{-13} & 1 \\ 0 & 1 - 10^{13} \end{bmatrix}$$

```
1  function A = LU(A, n)
2  % A = LU(A, n)
3  % Questo algoritmo restituisce nella matrice di ingresso la
       fattorizzazione
4  % della matrice stessa di dimensione n.
5  % A = matrice da fattorizzare e successivamente fattorizzata
6  % n = dimensione della matrice da fattorizzare
7
8      p = [1 : n];
9      for i = 1 : n − 1
10         [mi, ki] = max(abs(A(i : n, i)));
11         if (mi == 0)
12             error(la matrice e' singolare);
13         end
14         ki = ki + i − 1;
15         if (ki > i)
16             A([i ki], :) = A([ki i], :);
17             p([i ki]) = p([ki i]);
18         end
19         A(i + 1 : n, i) = A(i + 1 : n, i) / A(i, i);
20         A(i + 1 : n, i + 1 : n) = A(i + 1 : n, i + 1 : n) − A(i + 1 : n, i
               ) ∗ A(i, i + 1 : n);
21     end
22 end
```

Calcoliamo b:

$$b = Ae = \begin{pmatrix} \epsilon & 1 \\ 1 & 1 \end{pmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} \approx 1 \\ 2 \end{bmatrix}$$

$$U\backslash(L\backslash b) = \begin{bmatrix} 0.992 \\ 1 \end{bmatrix}$$

$$A\backslash b = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Dai calcoli richiesti segue che :

$$\varepsilon_{U\backslash(L\backslash b)} = 5.6517 \cdot 10^{-4} \quad \text{ed} \quad \varepsilon_{A\backslash b} = 2.2204 \cdot 10^{-16}$$

Si nota che il vettore b calcolato col metodo di Gauss senza pivoting abbia un'accuratezza minore rispetto alla sua versione calcolata con il metodo di Gauss con pivoting.

Esercizio 3.7 *Scrivere una function Matlab specifica per la risoluzione di un sistema lineare con matrice dei coefficienti $A \in R^{n\times n}$ bidiagonale inferiore a diagonale unitaria di Toeplitz, specificabile con uno scalare α. Sperimentarne e commentarne le prestazioni (considerare il numero di condizionamento della matrice) nel caso in cui $n = 12$ e $\alpha = 100$ ponendo dapprima $b = (1, 101, \ldots, 101)^T$ (soluzione esatta $\hat{x} = (1, \ldots, 1)^T$) e quindi $b = 0.1 * (1, 101, \ldots, 101)^T$ (soluzione esatta $\hat{x} = (0.1, \ldots, 0.1)^T$).*

Soluzione: Ricordando che le matrici bidiagonali inferiori a diagonale unitaria di Toeplitz sono le matrici $A \in R^{n\times n}$ del tipo

$$\begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ \alpha & 1 & 0 & \cdots & 0 \\ 0 & \alpha & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \cdots & 0 & \alpha & 1 \end{pmatrix}$$

```
1  function [b] = sistema_Toeplitz(A, b)
2  % [b] = sistema_Toeplitz(A, b)
3  % Calcola la soluzione di Ax=b con A matrice bidiagonale inferiore a
4  % diagonale unitaria di Toeplitz
5  % A Matrice
6  % b vettore dei termini noti
7  % [b] soluzione
8  [n, m] = size(A);
9  if n~=m
10     error('matrice non quadrata');
11 end
12 for i=2:n
13     j=i-1
14     b(i) = b(i) - A(i,j)*b(j);
15     b(i) = b(i)/A(i,i);
16 end
17 end
18
19 function [A] = toeplitz_Generator(n, alfa)
20 % [A] = toeplitz_Generator(n, alfa)
21 % Genera una matrice Toeplitz
22 % n dimenstione matirce
23 % alfa valore della sottodiagonale
24 % [A] matrice di toeplitz
25 A(1,1) = 1;
26 for i=2:n
27     A(i,i-1) = alfa;
28     A(i,i) = 1;
29 end
30 end
```

Il seguente codice seguente applica le due function appena mostrate calcolando il condizionamento del problema e la soluzione nei due casi richiesti con $n = 12$ e $\alpha = 100$:

```
% Viene generata la matrice
A = toeplitz_Generator(12,100);
condizionamento = cond(A,2);

% primo caso
b = [1;101;101;101;101;101;101;101;101;101;101;101];
xPrimoCaso = sistema_Toeplitz(A,b);

% secondo caso
b = [1;101;101;101;101;101;101;101;101;101;101;101];
b = b * 0.1
xSecondoCaso = sistema_Toeplitz(A,b);
```

Il numero di conzionamento risulta essere infinito. Di seguito vengono mostrati i risultati ottenuti nel primo caso e nel secondo caso relative a x_1 e x_2:

$$x_1 = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}^T \qquad x_2 = \begin{bmatrix} 0.1000 \\ 0.1000 \\ 0.1000 \\ 0.1000 \\ 0.1000 \\ 0.1000 \\ 0.1000 \\ 0.1014 \\ -0.0407 \\ 14.1702 \\ -1.4069 \times 10^3 \\ 1.4070 \times 10^5 \end{bmatrix}^T$$

Esercizio 3.8 *Scrivere una function che, dato un sistema lineare sovradeterminato $Ax = b$, con $A \in R^{m \times n}$, $m > n$, $rank(A) = n$ e $b \in R^m$, preso come input b e l'output dell'Algoritmo 3.8 del libro (matrice A riscritta con la parte significativa dei vettori di Householder normalizzati con prima componente unitaria), ne calcoli efficientemente la soluzione nel senso dei minimi quadrati.*

Soluzione:

```
function [b] = soluzione_es_8(A, b)
    [m,n] = size(A);
    Q_t = eye(m);
    for i=1:n
        Q_t= [eye(i-1) zeros(i-1,m-i+1); zeros(i-1, m-i+1)' (eye(m-i+1)
           -(2/norm([1; A(i+1:m, i)], 2)^2)*([1; A(i+1:m, i)]*[1 A(i+1:m,
           i)']))]*Q_t;
    end
    b = sist_triang_sup(triu(A(1:n, :)), Q_t(1:n, :)*b);
end
```

Esercizio 3.9 *Inserire due esempi diutilizzo della function implementata per il punto 8 e confrontare la soluzione ottenuta con quella fornita dal comando $A\backslash b$.*
Soluzione: Per il primo esempio utilizzeremo:

$$A_1 = \begin{pmatrix} 4 & 2 & 2 \\ 1 & 1 & 2 \\ 2 & 3 & 4 \\ 2 & 1 & 1 \end{pmatrix} \qquad b_1 = \begin{bmatrix} 4 \\ 5 \\ 4 \\ 1 \end{bmatrix}$$

La funzione restituisce:

$$x_1 = \begin{bmatrix} 1.533333333333334 \\ -5.99999999999999 \\ 4.73333333333333 \end{bmatrix}^T \qquad e \qquad A_1\backslash b_1 = \begin{bmatrix} 1.53333333333333 \\ -6.00000000000000 \\ 4.73333333333333 \end{bmatrix}^T$$

Per il secondo esempio invece:

$$A_1 = \begin{pmatrix} 1 & 2 \\ 2 & 6 \\ 4 & 3 \end{pmatrix} \qquad b_1 = \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix}$$

La funzione restituisce:

$$x_1 = \begin{bmatrix} 1.15014164305949 \\ 0.532577903682720 \end{bmatrix}^T \qquad e \qquad A_1\backslash b_1 = \begin{bmatrix} 1.15014164305949 \\ 0.532577903682719 \end{bmatrix}^T$$

In entrambi i casi $A\backslash b \approx x$
Il codice Matlab:

```
% primo esempio
A1 = [4 2 2; 1 1 2; 2 3 4; 2 1 1];
b1= [4 5 4 1]';

x1 = soluzione_es_8(algoritmo_3_8(A1), b1);
x1_Ab = A1\b1;

% secondo esempio
A2 = [1 2; 2 6; 4 3];
b2 = [4 5 6]';
x2 = soluzione_es_8(algoritmo_3_8(A2), b2);
x2_Ab = A2\b2;

function [b] = soluzione_es_8(A, b)
    [m,n] = size(A);
    Qt = eye(m);
    for i=1:n
        Qt= [eye(i-1) zeros(i-1,m-i+1); zeros(i-1, m-i+1)' (eye(m-i+1)-(2/
           norm([1; A(i+1:m, i)], 2)^2)*([1; A(i+1:m, i)]*[1 A(i+1:m, i)
           '']))]*Qt;
    end
    b = sistema_triang_sup(triu(A(1:n, :)), Qt(1:n, :)*b);
end
```

```
22
23 % fattorizzazione QR di householder
24 function A = algoritmo_3_8(A)
25     [m,n] = size(A);
26     for i=1:n
27         alpha = norm(A(i:m, i));
28         if alpha==0
29             error(il rango non e' massimo)
30         end
31         if A(i,i)>=0
32             alpha = -alpha;
33         end
34         v = A(i,i) - alpha;
35         A(i,i) = alpha;
36         A(i+1:m,i) = A(i+1:m,i)/v;
37         beta = -v/alpha;
38         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));
39     end
40 end
41
42 function [b] = sistema_triang_sup(A, b)
43         for j=length(A):-1:1
44                 b(j)=b(j)/A(j,j);
45                 for i=1:j-1
46                         b(i)=b(i)-A(i,j)*b(j);
47                 end
48         end
49 end
```

Esercizio 3.10 *Scrivere una function Matlab che realizza il metodo di Newton per un sistema nonlineare (prevedere un numero massimo di iterazioni e utilizzare il criteri di arresto basato sull'incremento in norma euclidea). Utilizzare la function costruita al punto 4 per la risoluzione del sistema lineare ad ogni iterazione.*

Soluzione:

```
1  function [x] = sistemaNonLineare(F, J, b, imax, tolx)
2  % [x] = sistemaNonLineare(F, J, imax, tolx)
3  % Applica il metodo di Newton per un sistema non lineare.
4  % F sistema non lineare
5  % J matrice Jacobiana di F
6  % b vettore dei termini noti
7  % imax numero massimo di iterazioni
8  % tolx tolleranza
9  % [x] soluzione
10 i = 0;
11 xold = 0;
12 x = b;
13 while (i < imax) && (norm(x-xold) > tolx)
```

```
    i = i+1;
    xold = x;
    [A, p] = fatt_LUpivot(feval(J,x));
    x = x+sistema_LUpivot(A, p, -feval(F,x));
end
end
```

Esercizio 3.11 *Verificato che la funzione $f(x_1, x_2) = x_1^2 + x_2^3 - x_1x_2$ ha un punto di minimo relativo in (1/12, 1/6), costruire una tabella in cui si riportano il numero di iterazioni eseguite, e la norma euclidea dell'ultimo incremento e quella dell'errore con cui viene approssimato il risultato esatto utilizzando la function sviluppata al punto precedente per valori delle tolleranze pari a 10^{-t}, con $t = 3{,}6$. Utilizzare (1/2, 1/2) come punto di innesco. Verificare che la norma dell'errore è molto più piccola di quella dell'incremento (come mai?)*

Soluzione: Verifichiamo l'esistenza di un punto di minimo relativo in $(\frac{1}{12}, \frac{1}{6})$ considerando il sistema non lineare:

$$F(\vec{x}) = \vec{0} \quad \text{con} \quad F = \begin{bmatrix} \frac{\partial}{\partial x_1} f(x_1, x_2) \\ \frac{\partial}{\partial x_2} f(x_1, x_2) \end{bmatrix}$$

$$\frac{\partial}{\partial x_1} f(x_1, x_2) = 2x_1 - x_2 \quad \frac{\partial}{\partial x_2} f(x_1, x_2) = 3x_2^2 - x_1$$

$$\begin{cases} 2x_1 - x_2 = 0 \\ 3y^2 - x_1 = 0 \end{cases} \quad \text{ha come soluzioni} \quad \begin{bmatrix} 0 & 0 \end{bmatrix} \quad \text{e} \quad \begin{bmatrix} \frac{1}{12} & \frac{1}{6} \end{bmatrix}$$

I punti trovati sono punti stazionari della funzione data.
Consideriamo la matrice Hessiana della funzione $f(x, x_2)$, che coincide con la matrice Jacobiana della funzione F:

$$H = \begin{bmatrix} f_{x_1x_1} & f_{x_1x_2} \\ f_{x_2x_1} & f_{x_2x_2} \end{bmatrix} = \begin{bmatrix} 2 & -1 \\ -1 & 6x_2 \end{bmatrix} = J_F \quad \det(H) = 12x_2 - 1$$

Il determinante della matrice Hessiana è positivo per $x_2 = \frac{1}{6}$, ed il primo elemento è positivo qui abbiamo un punto di minimo relativo in $(\frac{1}{12}, \frac{1}{6})$.

I seguenti dati sono stati ottenuti considerando come soluzione esatta $\hat{x} = [\frac{1}{12}, \frac{1}{6}]$ e come ultimo incremento la quantità $||x^{(i)} - x^{(i-1)}||$:

Tolleranza	iterazioni	$\|\|x^{(i)} - x^{(i-1)}\|\|$	$\|\|x - \hat{x}\|\|$
10^{-3}	5	2.8369×10^{-4}	4.3190×10^{-7}
10^{-4}	6	4.3190×10^{-7}	1.0011×10^{-12}
10^{-5}	6	4.3190×10^{-7}	1.0011×10^{-12}
10^{-6}	6	4.3190×10^{-7}	1.0011×10^{-12}

La norma dell'ultimo incremento sara' molto minore rispetto alla norma dell'errore di approssimanzione del risultato. Avendo che il metodo di Newton ha ordine di convergenza pari a 2, che consente all'approssimazione del risultato di convergere velocemente verso il risultato esatto.

```
x = [1/2, 1/2]';
tolx = 10^-3;

F = @(x) [2*x(1) - x(2); 3*x(2)^2 - x(1)];
J = @(x) [2, -1; -1, 6*x(2)];

[x, i, in, er] = nonLineare_newtonMod(F, J, x, 500, tolx);

function [x, i, in, er] = nonLineare_newtonMod(F, J, x, imx, tolx)
    i = 0;
    xold = x+100;
    while (i < imx) && (norm(x-xold) > tolx)
        i = i+1;
        xold = x;
        [A, p] = fatt_LUpivot(feval(J,x));
        x = x+sistema_LUpivot(A, -feval(F,x), p);
    end
    er = norm(x-[1/12, 1/6]');
    in = norm(x-xold);
end
```

Capitolo 4

Esercizi A.A. 2017-18

Esercizio 4.1 *Scrivere una function Matlab che implementi il calcolo del polinomio interpolante di grado n in forma di Lagrange.*
La forma della function deve essere del tipo: `Capitolo4/y = lagrange( xi, fi, x )`
Soluzione:

```
function y = lagrange(xi, fi, x)
% function y = lagrange( xi, fi, x )
% xi vettore dei punti di ascissa
% fi vettore dei valori di f(x)
% x vettore di punti in cui calcolare f(x)
% y valore di f(x)
if length(xi) ~= length(fi)
    error('xi e fi hanno lunghezza diversa, deve essere uguale')
end
n = length(xi)-1;
m = length(x);
y = zeros(size(x));
for i=1:m
    for j=1:n+1
        p = 1;
        for k=1:n+1
            if j~=k
                p = p*(x(i)-xi(k))/(xi(j)-xi(k));
            end
        end
        y(i) = y(i)+fi(j)*p;
    end
end
return
end
```

Esercizio 4.2 *Scrivere una function Matlab che implementi il calcolo del polinomio interpolante di grado n in forma di Newton.*
La forma della function deve essere del tipo: `Capitolo4/y = newton( xi, fi, x )`
Soluzione:

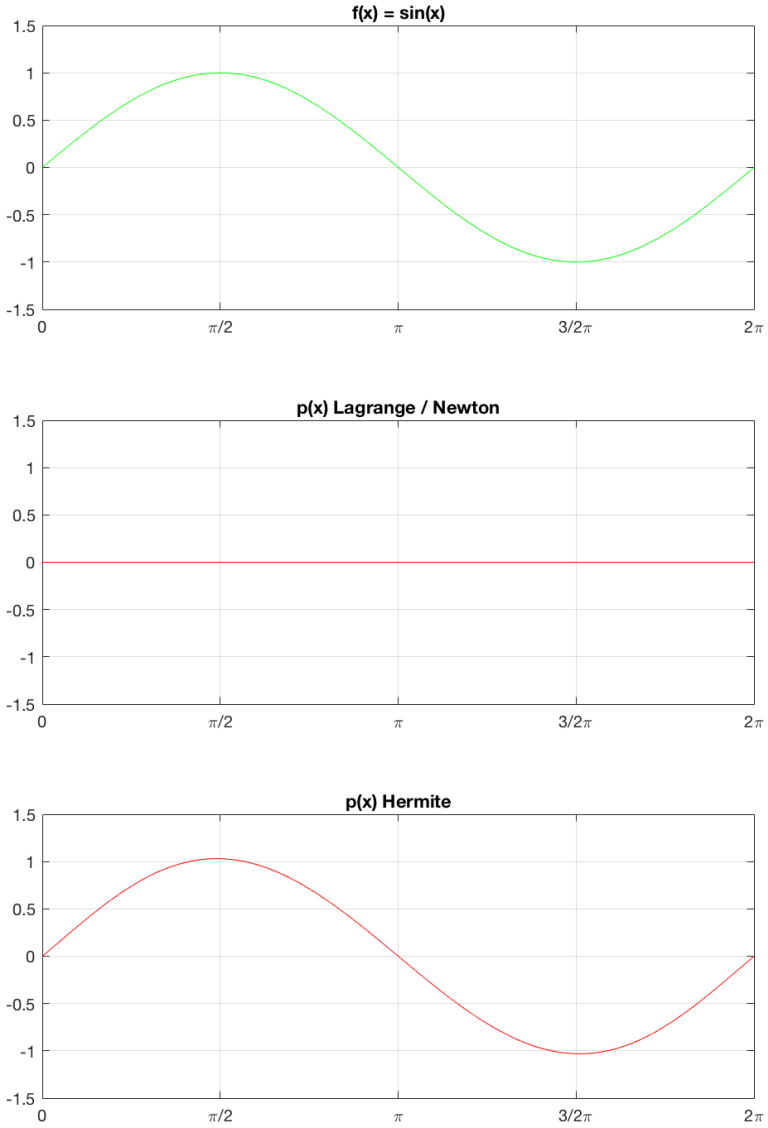
```
function y = newton(xi,fi,x)
% function y = newton(xi,fi,x)
% Implementa il calcolo del polinomio interpolante di grado n in forma di
   Newton
%  xi vettore delle ascisse di interpolazione
%  fi vettore dei valori della funzione in x
%  x vettore dei punti in cui valutare il polinomio
%  y vettore dei valori del polinomio valutato sui punti x.
if length(xi) ~= length(fi)
    error('xi e fi hanno lunghezza diversa!')
end
dd = diff_div(xi, fi);
y = dd(length(dd));
for k = length(dd)-1:-1:1
    y = y*(x-xi(k))+dd(k);
end
end

function [fi] = diff_div(xi, fi)
for i = 1:length(xi) - 1
    for j = length(xi):-1:i+1
        fi(j) = (fi(j) - fi(j-1))/(xi(j)-xi(j-i));
    end
end
end
```

Esercizio 4.3 *Scrivere una function Matlab che implementi il calcolo del polinomio interpolante di Hermite.*
La forma della function deve essere del tipo:
`Capitolo4/y = hermite( xi, fi, f1i, x )`
Soluzione:

```
function y = hermite(xi, fi, f1i, x)
% function y = hermite(xi, fi, f1i, x)
%   xi vettore delle ascisse
%   fi vettore delle valutazioni di f(x)
%   f1i vettore delle valutazioni di f'(x)
%   x punto da valutare
%   y vettore riscrito con le differenze divise
if length(xi) ~= length(fi) || length(f1i) ~= length(fi)
    error('xi, fi e f1x devono avere la stessa lunghezza')
end
% combino opportunamente i vettori
xih = zeros(length(xi)*2, 1);
fih = zeros(length(fi)*2, 1);
for i = 1:length(xi)
    xih(i+i-1) = xi(i);
    xih(i+i) = xi(i);
    fih(i+i-1) = fi(i);
```

```
    fih(i+i) = f1i(i);
end
ddh = diffDiviseHermite(xih, fih);
y = ddh(1);
for i = 2 : length(dd)
    prod = ddh(i);
    for j=1:i-1
        prod = prod*(x-xih(j));
    end
    y = y + prod;
end
end

function [fi] = diffDiviseHermite(xi, fi)
%  function [f] = diffDiviseHermite(x, f)
%  xi vettore delle ascisse
%  fi vettore con f(x0), f'(x0),..., f(xn), f'(xn)
%  fi vettore riscrito con le differenze divise
    n = length(xi)-1;
    for i = n:-2:3
        fi(i) = (fi(i)-fi(i-2))/(xi(i)-xi(i-1));
    end
    for j = 2:n
        for i = n+1:-1:j+1
            fi(i) = (fi(i)-fi(i-1))/(xi(i)-xi(i-j));
        end
    end
return
end
```

Esercizio 4.4 *Utilizzare le functions degli esercizi precedenti per disegnare l'approssimazione della funzione* $\sin(x)$ *nell'intervallo* $[0, 2\pi]$*, utilizzando le ascisse di interpolazione* $x_i = i\pi$*,* $i = 0, 1, 2$*.*
Soluzione:



Possiamo notare come il polinomio di Lagrange e Newton genera una retta $y = 0$ essendo $f_i = 0$ in tutte le ascisse.

```
% funzione esatta
f = @(x) sin(x);
f1 = @(x) cos(x);
interval = [0, 2*pi];

% polinomi interpolanti, function es 1, 2, 3
xi = [0, pi, 2*pi];
```

```
fi = [f(0), f(pi), f(2*pi)];
f1i = [f1(0), f1(pi), f1(2*pi)];

pn = @(x) newton(xi, fi, x); %Lagrange genererÃ  lo stesso polinomio
ph = @(x) hermite(xi, fi, f1i, x);

% plots
subplot(3,1,1)
fplot(f, interval, 'g')
grid on
title('f(x) = sin(x)')
xlim([0, 2*pi])
xticks([0 pi/2 pi pi+pi/2 2*pi])
xticklabels({'0','\pi/2','\pi','3/2\pi','2\pi'})
ylim([-1.5, 1.5])

subplot(3,1,2)
fplot(pn, interval, 'r')
grid on
title('p(x) Lagrange / Newton')
xlim([0, 2*pi])
xticks([0 pi/2 pi pi+pi/2 2*pi])
xticklabels({'0','\pi/2','\pi','3/2\pi','2\pi'})
ylim([-1.5, 1.5])

subplot(3,1,3)
fplot(ph, interval, 'r')
grid on
title('p(x) Hermite')
xlim([0, 2*pi])
xticks([0 pi/2 pi pi+pi/2 2*pi])
xticklabels({'0','\pi/2','\pi','3/2\pi','2\pi'})
ylim([-1.5, 1.5])
```

Esercizio 4.5 *Scrivere una function Matlab che implementi la spline cubica interpolante (naturale o not-a-knot, come specificato in ingresso) delle coppie di dati assegnate. La forma della funcion deve essere del tipo:* `Capitolo4/y = spline3(xi, fi, x, tipo)`
Soluzione: Il seguente codice Matlab implementa la function richiesta. Per rendere il codice più leggibile sono state craete varie sottofunzioni.
Il codice è stato testato con un banale esempio:

```
% esempio con f(x) = (pi*x)/(x+1)
xi = [0,1,2,3];
fi = [0, 1.570796, 2.094395, 2.3561944];
x = 1.5;

% not-a-knot
y_nan = spline3(xi, fi, x, true);
% naturale
```

```
y_nat = spline3(xi, fi, x, false);
y = (pi*x)/(x + 1);


function y = spline3(xi, fi, x, isNotAKnot)
% function y = spline3(xi, fi, x, isNotAKnot)
% xi vettore delle ascisse
% fi vettore delle valutazioni di f(x)
% punto da valutare
% isNotAKnot true se not-a-knot, false se naturale
% y risultato approssimazione di f(x)
    s = p_spline3(xi, fi, isNotAKnot);
    n = 0;
    for i = 1:xi(length(xi))
        if x > xi(i) && x <= xi(i+1)
            n = i;
            break
        end
    end

    y = double(subs(s(n), x));
end

function s = p_spline3(xi, fi, tipo)
    n = length(xi) - 1;
    xis = zeros(1, n - 1);
    phi = zeros(1, n - 1);
    for i = 1 : n - 1
        phi(i) = ( xi(i + 1) - xi(i) ) / ( xi(i + 2) - xi(i) );
        xis(i) = ( xi(i + 2) - xi(i + 1) ) / ( xi(i + 2) - xi(i) );
    end
    dd = diff_div_spline3(xi, fi);
    if tipo
        m = vettore_sistema_spline3(phi, xis, dd);
    else
        m = sistema_spline3(phi, xis, dd);
    end

    s = espressione_spline3(xi, fi, m);
end

function fi = diff_div_spline3(xi, fi)

    n = length(xi) - 1;

    for j = 1 : 2
        for i = n + 1 : - 1 : j + 1
            fi(i) = ( fi(i) - fi(i - 1) )/(xi(i) - xi(i - j) );
```

```
57          end
58      end
59
60      fi = fi(3 : length(fi))';
61  end
62
63  function m = sistema_spline3(phi, xi, dd)
64      n = length(xi) + 1;
65      u = zeros(1, n - 1);
66      l = zeros(1, n - 2);
67      u(1) = 2;
68      for i = 2 : n - 1
69          l(i) = phi(i) / u(i - 1);
70          u(i) = 2 - l(i) * xi(i - 1);
71      end
72      dd = 6 * dd;
73      y = zeros(1, n - 1);
74      y(1) = dd(1);
75      for i = 2 : n - 1
76          y(i) = dd(i) - l(i) * y(i - 1);
77      end
78      m = zeros(1, n - 1);
79      m(n - 1) = y(n - 1) / u(n - 1);
80      for i = n - 2 : - 1 : 1
81          m(i) = (y(i) - xi(i) * m(i + 1)) / u(i);
82      end
83      m = [0 m 0];
84  end
85
86  function m = vettore_sistema_spline3(phi, xi, dd)
87      n = length(xi) + 1;
88      if n + 1 < 4
89          error('Not A Knot con meno di 4 ascisse!');
90      end
91      dd = [6 * dd(1); 6 * dd; 6 * dd(length(dd))];
92      w = zeros(n, 1);
93      u = zeros(n + 1, 1);
94      l = zeros(n, 1);
95      y = zeros(n + 1, 1);
96      m = zeros(n + 1, 1);
97      u(1) = 1;
98      w(1) = 0;
99      l(1) = phi(1);
100     u(2) = 2 - phi(1);
101     w(2) = xi(1) - phi(1);
102     l(2) = phi(2) / u(2);
103     u(3) = 2 - ( l(2) * w(2) );
104     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


function s = espressione_spline3(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
```

Esercizio 4.6 *Scrivere una function Matlab che implementi il calcolo delle ascisse di Chebyshev per il polinomio interpolante di grado n, su un generico intervallo $[a, b]$. La function deve essere del tipo:* `Capitolo4/ xi = ceby( n, a, b )`

Soluzione:

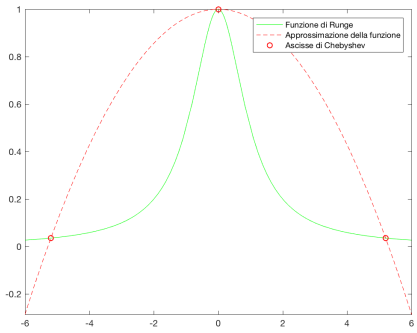
```
function xi = ceby(n, a, b)
% function xi = ceby(n, a, b)
% n numbero di ascisse da cercare
% intervallo da a a b
% xi vettore con le ascisse cercate di Chebyshev
  xi = zeros(n+1, 1);
    for i = 0:n
      xi(n+1-i) = (a+b)/2 + cos(pi*(2*i+1)/(2*(n+1)))*(b-a)/2;
```

```
9       end
10  end
```

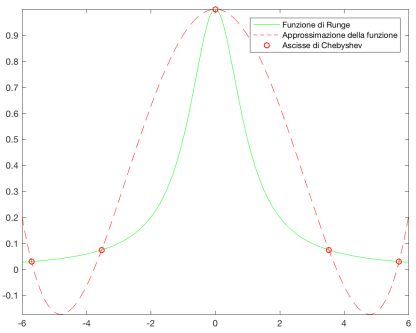
Esercizio 4.7 *Utilizzare le function degli Esercizi 4.1 e 4.6 per graficare l'approssimazione della funzione di Runge sull'intervallo* $[-6, 6]$*, per* $n = 2, 4, 6, \ldots, 40$*. Stimare numericamente l'errore commesso in funzione del grado* n *del polinomio interpolante.*

Soluzione: Di seguito i grafici che mostrano i polinomi interpolanti di grado n calcolati utilizzando come punti di interpolazione quelli corrispondenti alle n ascisse di Chebyshev, sovrapposti al grafico della funzione di Runge: $f(x) = \frac{1}{1+x^2}$.

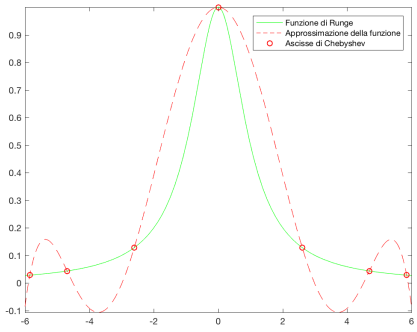
$n = 2$



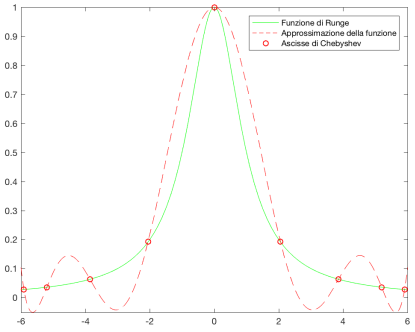
$n = 4$



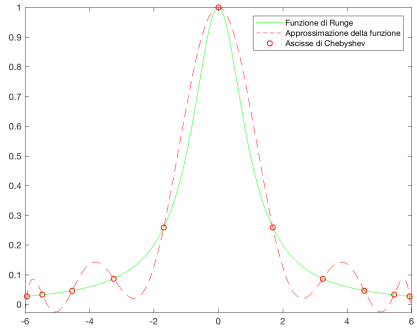
$n = 6$



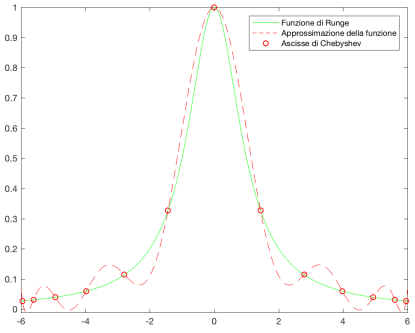
$n = 8$



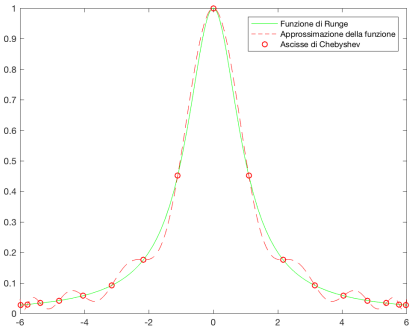
$n = 10$



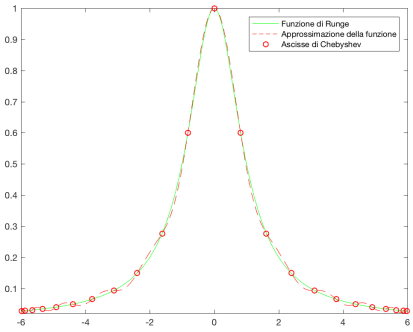
$n = 12$



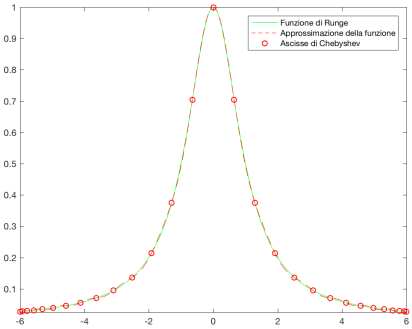
$n = 16$



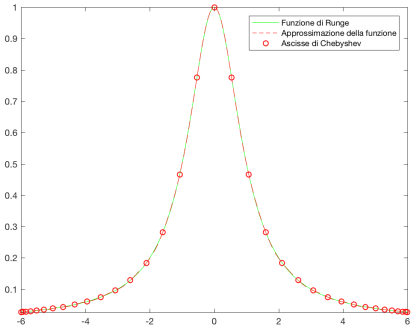
$n = 22$



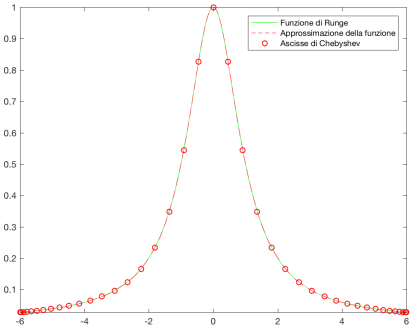
$n = 28$



$n = 30$



$n = 40$



L'errore è stato calcolato seguendo la seguente formula:

$$||e|| \approx ||f(x) - p_n(x)||_{\text{inf}}$$

Dove f è intesa come la funzione di Runge e p il suo polinomio interpolante.

L'errore stimato è visibile nella seguente tabella:

n	errore
2	0.6577
4	0.4741
6	0.3371
8	0.2365
10	0.1640
12	0.1129
14	0.0772
16	0.0534
18	0.0399
20	0.0295
22	0.0216
24	0.0157
26	0.0113
28	0.0081
30	0.0058
32	0.0041
34	0.0029
36	0.0021
38	0.0015
40	0.0011

Notiamo che, utilizzando le ascisse di Chebyshev, aumentando il numero di punti otteniamo un'approssiamazione sempre più vicina alla funzione di Runge. Infatti l'errore diminuisce all'aumentare di n tendendo a 0 quando n tende all'infinito.

Esercizio 4.8 *Relativamente al precedente esercizio, stimare numericamente la crescita della costante di Lebesgue.*
Soluzione: La stima della costante di Lebesgue mediante le ascisse di Chebyshev è data dalla seguente formula:

$$\Lambda_n \approx \frac{2}{\pi} \log n$$

Ci si aspetta quindi che abbia una crescita logaritmica al crescere di n.

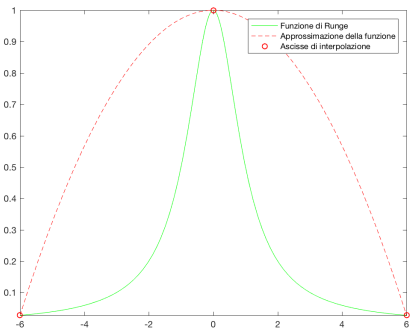
La tebella seguente mette in evidenza tale comportamento:

n	lebesgue
2	0.4413
4	0.8825
6	1.1407
8	1.3238
10	1.4659
12	1.5819
14	1.6801
16	1.7651
18	1.8401
20	1.9071
22	1.9678
24	2.0232
26	2.0742
28	2.1213
30	2.1653
32	2.2064
34	2.2450
36	2.2813
38	2.3158
40	2.3484

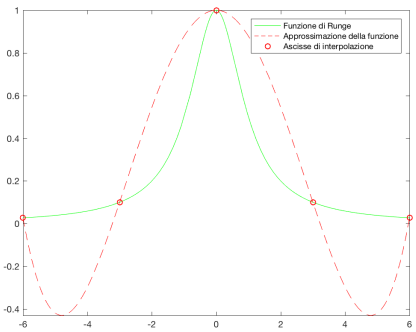
Esercizio 4.9 *Utilizzare la function ell'Esercizio 4.1 per approssimare la funzione di Runge sull'intervallo* $[-6, 6]$*, su una partizione uniforme di* $n+1$ *ascisse per* $n = 2, 4, 6, \ldots, 40$*. Stimare le corrispondenti costanti di Lebesgue.*

Soluzione:

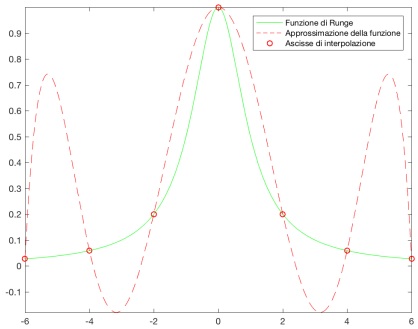
$n = 2$



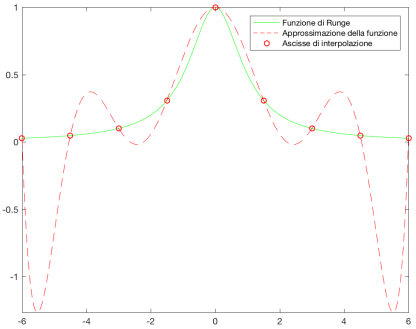
$n = 4$



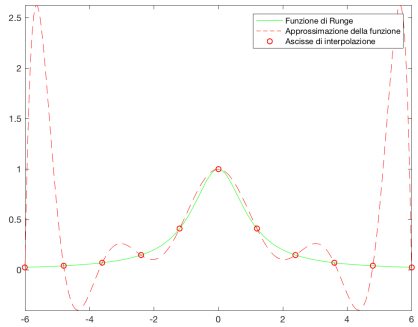
$n = 6$



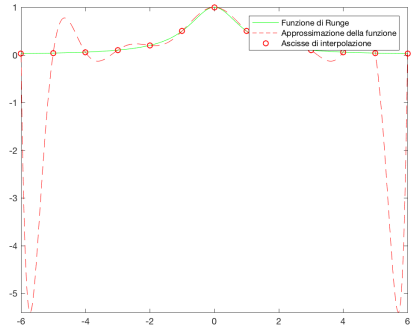
$n = 8$



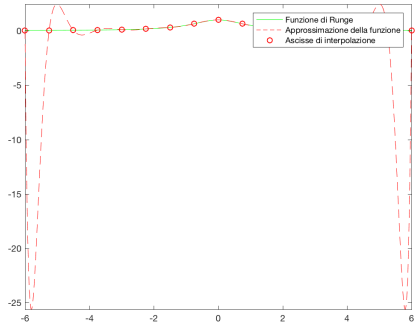
$n = 10$



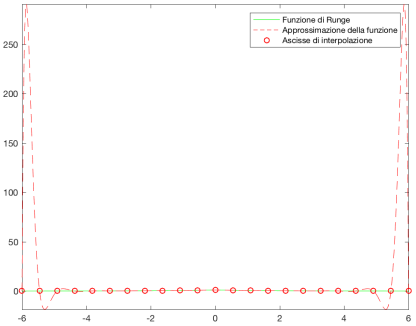
$n = 12$



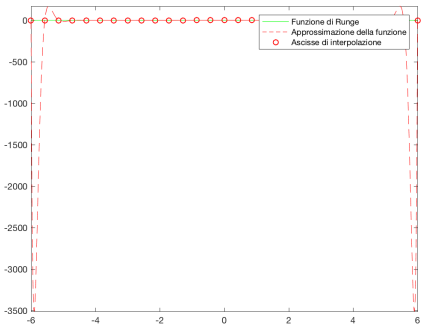
$n = 16$



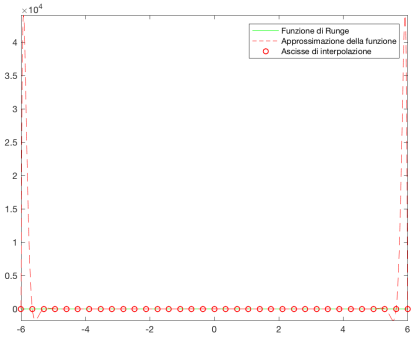
$n = 22$



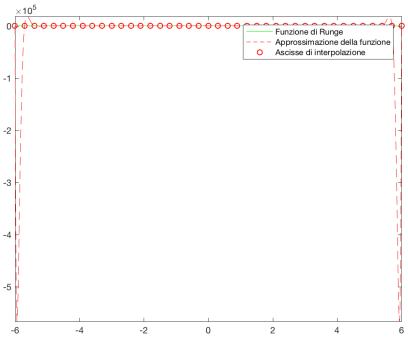
$n = 28$



$n = 34$



$n = 40$



Di seguito riportiamo la tabella con le stime degli errori e della costante di Lebesgue in funzione del grado n:

n	errore
2	0.6577
4	0.4741
6	0.3371
8	0.2365
10	0.1640
12	0.1129
14	0.0772
16	0.0534
18	0.0399
20	0.0295
22	0.0216
24	0.0157
26	0.0113
28	0.0081
30	0.0058
32	0.0041
34	0.0029
36	0.0021
38	0.0015
40	0.0011

Esercizio 4.10 *Stimare, nel senso dei minimi quadrati, posizione, velocità iniziale ed accelerazione relative ad un moto rettilineo uniformemente accelerato per cui sono note le seguenti misurazioni dele coppie (tempo, spazio):*
$(1, 2.9)$ $(1, 3.1)$ $(2, 6.9)$ $(2, 7.1)$ $(3, 12.9)$ $(3, 13.1)$ $(4, 20.9)$ $(4, 21.1)$ $(5, 30.9)$ $(5, 31.1)$
Soluzione: Il problema posto consiste nel risolvere

$$y = s(t) = x_0 + v_0 t + \frac{1}{2} a t^2, \qquad \text{con } a, x_0 \text{ e } v_0 \text{ costanti}$$

La stima, nel senso dei minimi quadrati, equivale alla risoluzione del sistema lineare sovradeterminato Ax=b:

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \\ 1 & 3 & 9 \\ 1 & 4 & 16 \\ 1 & 4 & 16 \\ 1 & 5 & 25 \\ 1 & 5 & 25 \end{pmatrix} \begin{pmatrix} x \\ v \\ a/2 \end{pmatrix} = \begin{pmatrix} 2.9 \\ 3.1 \\ 6.9 \\ 7.1 \\ 12.9 \\ 13.1 \\ 20.9 \\ 21.1 \\ 30.9 \\ 31.1 \end{pmatrix}$$

Il problema è ben posto ed esiste un'unica soluzione in quanot abbiamo misurazioni in $n + 1$ ascisse.
La matrice A è una matrice di Vandermonde con rango massimo 3.
É possibile calcolare una soluzione per il sistema dato utilizzando le function per la risoluzione e fattorizzazione dei sistemi QR sviluppati negli eserci precedenti:

$$\begin{pmatrix} x \\ v \\ a/2 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 0.99999 \end{pmatrix}$$

Capitolo 5

Esercizi A.A. 2017-18

Esercizio 5.1 *Scrivere una function Matlab che implementi la formula composita dei trapezi su $n+1$ ascisse equidistanti nell'intervallo $[a, b]$ relativamente alla funzione implementata da* `Capitolo5/fun(x)`.
La function deve essere del tipo `Capitolo5/If = trapcomp(a, b, fun, tol)`.
Soluzione:

```
function If = trapcomp(n, a, b, fun)
% function If = trapcomp(n , a, b, fun)
% Calcola l'integrale della funzione nell'intervallo a b,
% utilizzando la formula dei trapezi composita.
% fun funzione integranda
% If valore approssimato dell'integrale definito della funzione
If = 0;
h = (b-a) / n;
for i = 1:n-1
    If = If + fun(a + i*h);
end
If = (h/2) * (2*If + fun(a) + fun(b));
end
```

Esercizio 5.2 *Scrivere una function Matlab che implementi la formula composita di Simpson su $2n+1$ ascisse equidistanti nell'intervallo $[a, b,]$ relativamente alla funzione implementata da* `Capitolo5/fun(x)`.
La function deve essere del tipo `Capitolo5/If = simpcomp(a, b, fun, tol)`.
Soluzione:

```
function If = simpcomp(n, a, b, fun)
% function If = trapcomp(n , a, b, fun)
% Calcola l'integrale della funzione nell'intervallo a b,
% utilizzando la formula di Simpson composita
% fun funzione integranda
% If valore approssimato dell'integrale definito della funzione
If = fun(a) - fun(b);
h = (b-a) / n;
for i = 1:n/2
    If = If + 4*fun(a+(2*i-1)*h) + 2*fun((a+2*i*h));
```

```
11 end
12 If = If*(h/3);
13 end
```

Esercizio 5.3 *Scrivere una function Matlab che implementi la formula composita dei trapezi adattativa nell'intervallo* $[a, b]$ *relativamente alla funzione implementata da* `Capitolo5/fun(x)` *e con tolleranza* `Capitolo5/tol`.
La function deve essere del tipo `Capitolo5/If = trapad(a, b, fun, tol)`.
Soluzione:

```
1  function If = trapad(a, b, fun, tol)
2  % function If = trapad(a, b, fun, tol)
3  % Calcola ricorsivamente l'integrale della funzione nell'intervallo a b
4  % utilizzando la formula dei trapezi adattiva.
5  % fun funzione integranda
6  % if approssimazione dell'integrale definito della funzione
7  h = (b-a)/2;
8  m = (b+a)/2;
9
10 If1 = h*(feval(fun, a) + feval(fun, b));
11 If = If1/2 + h*feval(fun, m);
12
13 err = abs(If - If1)/3;
14
15 if err > tol
16     iSx = trapad(a, m, fun, tol/2);
17     iDx = trapad(m, b, fun, tol/2);
18
19     If = iSx + iDx;
20 end
21 end
```

Esercizio 5.4 *Scrivere una function Matlab che implementi la formula composita di Simpson adattativa nell'intervallo* $[a, b]$ *relativamente alla funzione implementata da* `Capitolo5/fun(x)` *e con tolleranza* `Capitolo5/tol`.
La function deve essere del tipo `Capitolo5/If = simpad(a, b, fun, tol)`.
Soluzione:

```
1  function If = simpad(a, b, fun, tol)
2  % function If = simpad(a ,b ,fun, tol)
3  % Calcola l'integrale della funzione nell'intervallo a b utilizzando la
4  % formula di simpson adattiva
5  % fun funzione integranda
6  % If approssimazione dell'integrale definito della funzione
7  h = (b-a) / 6;
8  m = (a+b) / 2;
9  m1 = (a+m) / 2;
10 m2 = (m+b) / 2;
11
```

```
If1 = h*(feval(fun, a) + 4*feval(fun, m) + feval(fun, b));
If = If1/2 + h*(2*feval(fun, m1) + 2*feval(fun, m2) - feval(fun, m));

err = abs(If-If1) / 15;

if err>tol
    iSx = simpad(a, m, fun, tol/2);
    iDx = simpad(m, b, fun, tol/2);
    If = iSx+iDx;
end
end
```

Esercizio 5.5 *Calcolare quante valutazioni di funzione sono necessarie per ottenere una approssimazione di*

$$I(f) = \int_0^1 \exp(-10^6 x)dx$$

che vale 10^{-6} *in doppia precisione IEEE, con una tolleranza* 10^{-9}*, utilizzando le functions dei precedenti esercizi. Argomentare quantitativamente la risposta.*
Soluzione: Le functions degli esercizi precedenti esercizi sono state leggermente modificate e sono stati calcolati i seguenti valori:

Function	Valutazioni	Errore
Trapezi composita	10000001	8.3319×10^{-10}
Simpson composita	2000002	3.3715×10^{-10}
Trapezi adattiva	77823	1.1252×10^{-14}
Simpson adattiva	1038	1.6469×10^{-14}

Possiamo vedere che per la funzione presa in esame performano meglio le formule adattive in quanto individuano i nodi della partizione in base al comportamento locale della funzione, questo permette di minimizzare l'errore e di raggiungere la soglia di tolleranza prestabilita.
Viceversa le formule con ascisse equispaziate si sono rilevate inadeguate per questo tipo di funzione che presenta una rapida variazione di valore in una porzione dell'intervallo molto ristretta.
Il codice utilizzato per ottenere i risultati posti sopra:

```
% dati
If = 10^(-6);
fun = @(x) exp((-10^6)*x);
tol = 10^(-9);

[If_appr, nval] = trapcomp_val(10^7, 0, 1, fun);
print_res('Trapezi Composita', nval, abs(If-If_appr))

[If_appr, nval] = simpcomp_val(2*10^6, 0, 1, fun);
print_res('Simpson Composita', nval, abs(If-If_appr))

[If_appr, nval] = trapad_val(0, 1, fun, tol);
print_res('Trapezi Adattativa', nval, abs(If-If_appr))
```

```
14
15 [If_appr, nval] = simpad_val(0, 1, fun, tol);
16 print_res('Simpson Adattativa', nval, abs(If-If_appr))
17
18 % servono sempre n+1 valutazioni
19 function [If, nval] = trapcomp_val(n, a, b, fun)
20     If = 0;
21     nval = 0;
22     h = (b-a) / n;
23     for i = 1:n-1
24         If = If + fun(a + i*h);
25         nval = nval + 1;
26     end
27     If = (h/2) * (2*If + fun(a) + fun(b));
28     nval = nval + 2;
29 end
30
31 % servono sempre n+2 valutazioni
32 function [If, nval] = simpcomp_val(n, a, b, fun)
33     If = fun(a) - fun(b);
34     nval = 2;
35     h = (b-a) / n;
36     for i = 1:n/2
37         If = If + 4*fun(a+(2*i-1)*h) + 2*fun((a+2*i*h));
38         nval = nval + 2;
39     end
40     If = If*(h/3);
41 end
42
43 % tre valutazioni di fun ad ogni call
44 function [If, neval] = trapad_val(a, b, fun, tol)
45     h = (b-a)/2;
46     m = (b+a)/2;
47
48     If1 = h*(feval(fun, a) + feval(fun, b));
49     If = If1/2 + h*feval(fun, m);
50     neval = 3;
51
52     err = abs(If - If1) / 3;
53
54     if err > tol
55         [iSx, nSx] = trapad_val(a, m, fun, tol/2);
56         [iDx, nDx] = trapad_val(m, b, fun, tol/2);
57
58         If = iSx + iDx;
59         neval = neval + nSx + nDx;
60     end
61 end
```

```
62
63
64 % sei valutazioni di fun ad ogni call
65 function [If, neval] = simpad_val(a, b, fun, tol)
66     h = (b-a) / 6;
67     m = (a+b) / 2;
68     m1 = (a+m) / 2;
69     m2 = (m+b) / 2;
70
71     If1 = h*(feval(fun, a) + 4*feval(fun, m) + feval(fun, b));
72     If = If1/2 + h*(2*feval(fun, m1) + 2*feval(fun, m2) - feval(fun, m));
73
74     neval = 6;
75     err = abs(If-If1) / 15;
76
77     if err > tol
78         [iSx, nSx] = simpad_val(a, m, fun, tol/2);
79         [iDx, nDx] = simpad_val(m, b, fun, tol/2);
80
81         If = iSx+iDx;
82         neval = neval + nSx + nDx;
83     end
84 end
85
86 % function per stampare il risultato
87 function print_res(fun_name, nval, err)
88     disp(strcat(fun_name, ' - valutazioni f:'))
89     disp(nval)
90     disp(strcat(fun_name, ' - errore:'))
91     disp(err)
92 end
```

Capitolo 6

Esercizi A.A. 2017-18

Esercizio 6.1 *Scrivere una function Matlab che generi la matrice sparsa $n \times n$, con $n > 10$*

$$A_n = \begin{pmatrix} a_{11} & \dots & a_{1n} \\ \vdots & & \vdots \\ a_{n1} & \dots & a_{nn} \end{pmatrix}, \qquad con \qquad a_{ij} \begin{cases} 4 \ se \ i = j \\ -1 \ se \ i = j \pm 1 \\ -1 \ se \ i = j \pm 10 \end{cases}$$

Utilizzare a questo fine la function Matlab `spdiags`.
Soluzione:

```
function A = matrice_sparsa(n)
% function A = matrice_sparsa(n)
% Genera la matrice quadrata sparsa nxn con n maggiore di 10
% n numero di righe/colonne della matrice
% A matrice output
    if n <= 10
        error('n deve essere > 10')
    end

    ij = ones(n, 1) * 4;
    ij1 = ones(n, 1) * -1;
    ij10 = ij1;

    B = [ij ij1 ij1 ij10 ij10];
    d = [0, 1, -1, 10, -10];

    A = spdiags(B, d, n, n);
end
```

Esercizio 6.2 *Utilizzare il metodo delle potenze per calcolare l'autovalore dominante della matrice A_n del precedente esercizio, con una approssimazione $tol = 10^{-5}$, partendo da un vettore con elementi costanti. Riempire, quindi, la seguente tabella:*

n	*numero iterazioni effettuate*	*stima autovalore*
100		
⋮		
1000		

Soluzione: La tabella richiesta è la seguente:

n	iterazioni	autovalore
100	122	7.8200
200	270	7.8794
300	643	7.8912
400	395	7.8952
500	413	7.8975
600	499	7.8953
700	623	7.8958
800	432	7.8962
900	651	7.8974
1000	817	7.8947

Generata grazie al seguente codice matlab:

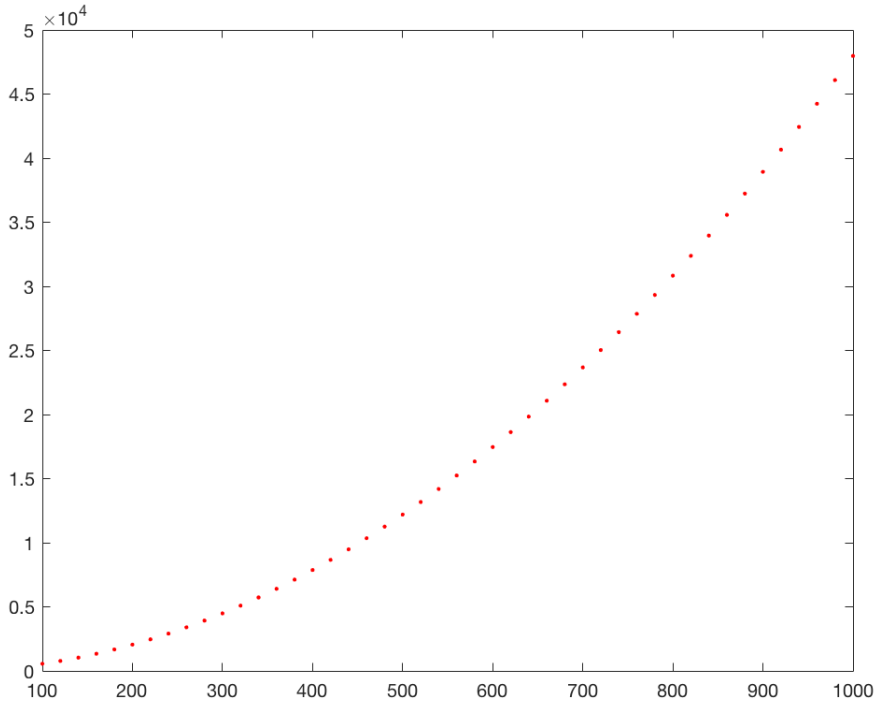
```
Table = cell2table(cell(0,3));
Table.Properties.VariableNames = {'n' 'iterazioni', 'autovalore'};

tol = 10^(-5);

for n = 100:100:1000
   [l, i] = potenze(matrice_sparsa(n), tol, ones(n,1));

   record = {n, i, l};
   Table = [Table; record];
end

uitable('Data',Table{:,:},'ColumnName',Table.Properties.VariableNames,...
    'RowName',Table.Properties.RowNames,'Units', 'Normalized', 'Position'
       ,[0, 0, 1, 1]);

function [lambda, i] = potenze(A, tol, x0, maxit)
% function [lambda, i] = potenze(A, tol, [x0, maxit])
% Restituisce l'autovalore dominante della matrice A e il numero di
% iterazioni necessarie per calcolarlo
% A matrice utilizzata per il calcolo
% tol tolleranza dell' approssimazione
% [x0] vettore di partenza
% [maxit] numero massimo di iterazioni
% lambda matrice quadrata nxn sparsa
% i numero di iterazioni
    [m,n] = size(A);
    if m ~= n
        error('La matrice deve essere quadrata.');
```

```
29      end
30      if x0(:)==0
31          error('Il vettore x0 non puo avere esclusivamente elementi nulli.'
                );
32      end % if da eliminare per calcolo con matrice
33
34      if nargin <= 2
35          x = rand(n, 1);
36      else
37          x = x0;
38      end
39      if nargin <= 3
40          maxit = 100*2*round(-log(tol));
41      end
42      x = x0; % da eliminare per calcolo con matrice
43      x = x / norm(x);
44      lambda = inf;
45      for i=1:maxit
46          lambda0 = lambda;
47          v = A * x;
48          lambda = x' * v;
49          err = abs(lambda - lambda0);
50          if err <= tol
51              break
52          end
53          x = v/norm(v);
54      end
55      if err > tol
56          warning(Raggiunto maxit);
57      end
58      return
59  end
```

Esercizio 6.3 *Utilizzare il metodo di Jacobi per risolvere il sistema lineare*

$$A_n \boldsymbol{x} = \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix},$$

dove A_n è la matrice definita in (1), con tolleranza tol $= 10^{-5}$, e partendo dal vettore nullo. Graficare il numero di iterazioni necessarie, rispetto alla dimensione n del problema, con n che varia da 100 a 1000 (con passo 20).
Soluzione:

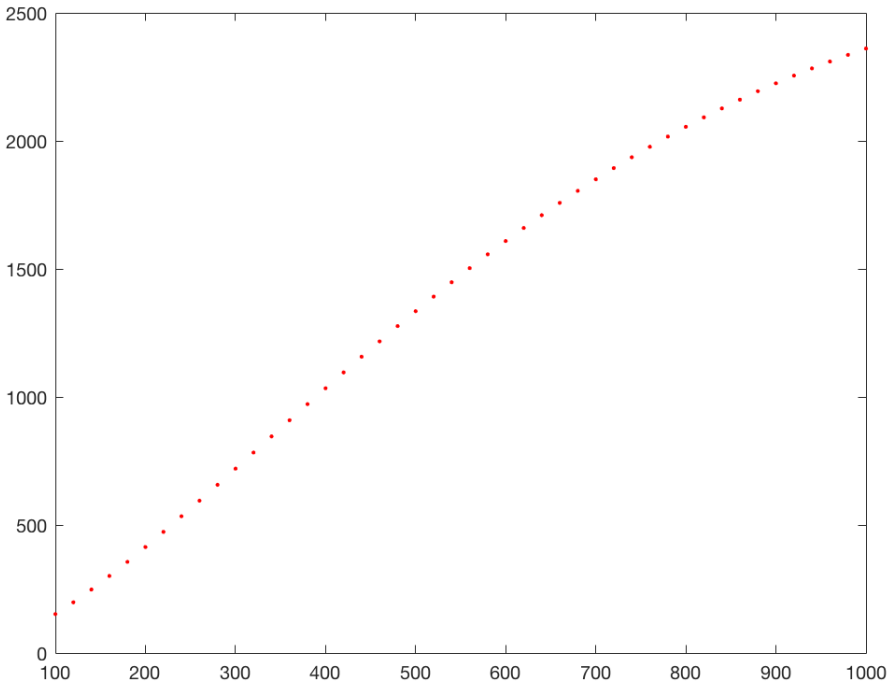


Il codice matlab che ha generato il grafico:

```
tol = 10^(-5);

for n = 100:20:1000
    [x, i] = jacobi(matrice_sparsa(n), ones(n, 1), zeros(n,1), tol);

    plot(n, i, 'r.')
    hold on
end

function [x,i] = jacobi(A, b, x0, tol)
% function [x,i] = jacobi(A, b, x0, tol)
% Restituisce la soluzione del sistema lineare Ax=b approssimata con il
% metodo di Jacobi e il numero di iterazioni eseguite.
% A matrice utilizzata per il calcolo
% b vettore dei termini noti
% tol tolleranza dell' approssimazione
```

```
% x0 vettore di partenza
% x soluzione approssimata del sistema

D = diag(diag(A));
J = -inv(D) * (A-D);
q = D \ b;
x = J *x0 + q;
i = 1;
err = norm(x-x0) / norm(x);

while err > tol
    x0 = x;
    x = J * x0 + q;
    err = norm(x-x0) / norm(x);
    i = i + 1;
end
end
```

Esercizio 6.4 *Ripetere una procedura analoiga a quella del precedente esercizio utilizzando il metodo di Gauss-Seidel.*
Soluzione:

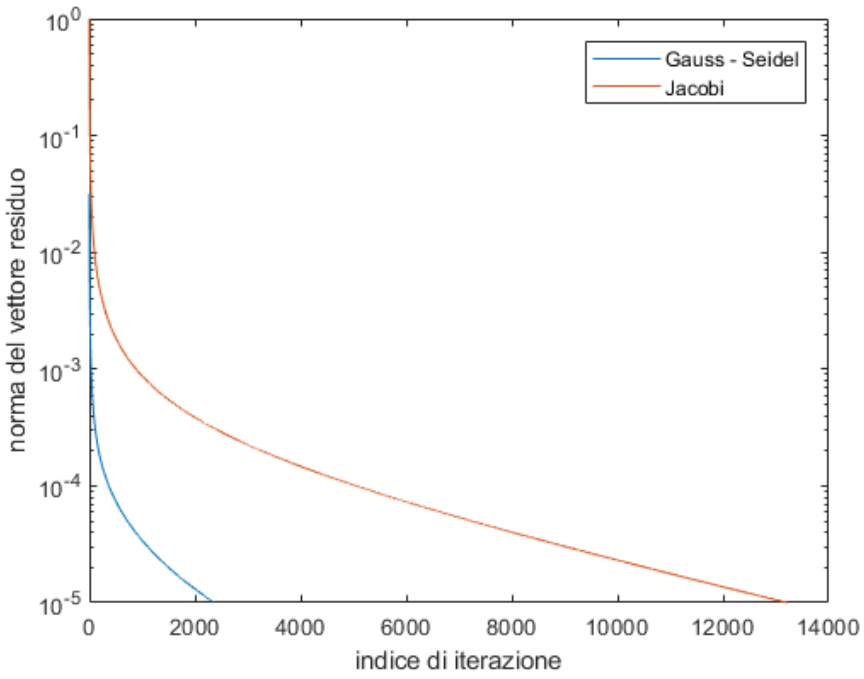


Il codice matlab che ha generato il grafico:

```
tol = 10^(-5);

for n = 100:20:1000
    [x, i] = gauss_seidel(matrice_sparsa(n), ones(n, 1), zeros(n,1), tol);
```

```
5
6       plot(n, i, 'r.')
7       hold on
8   end
9
10  function [x,i] = gauss_seidel(A, b, x0, tol)
11  % function [x,i] = jacobi(A, b, tol, [xo, maxit])
12  % Restituisce la soluzione del sistema lineare Ax=b approssimata con il
13  % metodo di Gauss–Seidel e il numero di iterazioni eseguite.
14  % A matrice utilizzata per il calcolo
15  % b vettore dei termini noti
16  % tol tolleranza dell' approssimazione
17  % x0 vettore di partenza
18  % x soluzione approssimata del sistema
19      D = diag(diag(A));
20
21      L = tril(A) - D;
22      U = triu(A) - D;
23
24      b1 = (D + L) \ b;
25
26      DI = inv(D + L);
27      GS = -DI * U;
28
29      x = GS * x0 + b1;
30      i = 1;
31      err = norm(x-x0, inf) / norm(x);
32
33      while err > tol
34          x0 = x;
35          x = GS * x0 + b1;
36          err = norm(x-x0, inf) / norm(x);
37          i = i + 1;
38      end
39  end
```

Esercizio 6.5 *Con riferimento al sistema lineare (2), con $n = 1000$, graficare la norma dei residui rispetto all'indice di iterazione, generati dai metodi di Jacobi e Gauss-Seidel. Utilizzare il formato* `semilogy` *per realizzare il grafico, corredandolo di opportune label.*
Soluzione: Il seguente grafico è realizzato con le ascisse in scala logaritmica tramite il formato semilogaritmico semilogy di Matlab:



Il codice matlab che ha generato il grafico:

```
tol = 10^(-5);
n = 1000;

A = matrice_sparsa(1000);
b = ones(n, 1);

[x_gs, i_gs, err_gs] = gauss_seidel_err(A, b, zeros(n, 1), tol);
[x_j, i_j, err_j] = jacobi_err(A, b, zeros(n,1), tol);

semilogy(1:i_gs, err_gs)
hold on
semilogy(1:i_j, err_j)
hold off

function [x,i, err] = gauss_seidel_err(A, b, x0, tol)
% function [x,i] = jacobi(A, b, tol, [xo, maxit])
% Restituisce la soluzione del sistema lineare Ax=b approssimata con il
% metodo di Gauss–Seidel e il numero di iterazioni eseguite.
% A matrice utilizzata per il calcolo
% b vettore dei termini noti
% tol tolleranza dell' approssimazione
% x0 vettore di partenza
```

```
% x soluzione approssimata del sistema
    D = diag(diag(A));

    L = tril(A) - D;
    U = triu(A) - D;

    b1 = (D + L) \ b;

    DI = inv(D + L);
    GS = -DI * U;

    x = GS * x0 + b1;
    i = 1;
    err(i) = norm(x-x0, inf) / norm(x);

    while err(i) > tol
        x0 = x;
        x = GS * x0 + b1;
        i = i + 1;
        err(i) = norm(x-x0, inf) / norm(x);
    end
end


function [x, i, err] = jacobi_err(A, b, x0, tol)

    D = diag(diag(A));
    J = - inv(D) * (A-D);

    q = D \ b;

    x = J*x0 + q;
    i = 1;
    err(i) = norm(x-x0)/norm(x);

    while err > tol
        x0 = x;
        x = J*x0 + q;
        i = i + 1;
        err(i) = norm(x-x0)/norm(x);
    end
end
```