

UNIVERSITÀ DEGLI STUDI DI FIRENZE
CORSO TRIENNALE IN INFORMATICA

Elaborato Calcolo Numerico

Autori

Alessandro De Cicco (matr.7009346)

Luca Fumagalli (matr.7004476)

Anno accademico: 2022/2023

Esercizio 1: Verificare che:

$$-\frac{1}{4}f(x-h) - \frac{5}{6}f(x) + \frac{3}{2}f(x+h) - \frac{1}{2}f(x+2h) + \frac{1}{12}f(x+3h) = hf'(x) + O(h^5)$$

Soluzione: Innanzitutto per semplificare i calcoli si può raccogliere:

$$\frac{1}{12}[-3f(x-h) - 10f(x) + 18f(x+h) - 6f(x+2h) + f(x+3h)] = hf'(x) + O(h^5)$$

Considerando un h sufficientemente piccolo si può considerare di approssimare la funzione con il polinomio di Taylor centrato in x_0 con la funzione:

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

Basterà approssimare il polinomio al quarto ordine:

$$\begin{aligned} f(x) &= f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0)^2 + \frac{f'''(x_0)}{3!}(x - x_0)^3 + \\ &\quad + \frac{f^{(4)}(x_0)}{4!}(x - x_0)^4 + O((x - x_0)^5) \end{aligned}$$

Da cui si può ricavare:

$$\begin{aligned} f(x-h) &= f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{6}f'''(x) + \frac{h^4}{24}f^{(4)}(x) + O(h^5) \\ f(x+h) &= f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(x) + \frac{h^4}{24}f^{(4)}(x) + O(h^5) \\ f(x+2h) &= f(x) + 2hf'(x) + 2h^2f''(x) + \frac{4}{3}h^3f'''(x) + \frac{2}{3}h^4f^{(4)}(x) + O(h^5) \\ f(x+3h) &= f(x) + 3hf'(x) + \frac{9}{2}h^2f''(x) + \frac{9}{2}h^3f'''(x) + \frac{27}{8}h^4f^{(4)}(x) + O(h^5) \end{aligned} \tag{1}$$

Quindi andando a sostituire si ottiene:

$$\begin{aligned} &\frac{-3f(x-h) - 10f(x) + 18f(x+h) - 6f(x+2h) + f(x+3h)}{12} = \\ &= \frac{(-3 - 10 + 18 - 6 + 1)f(x) + (3 + 18 - 12 + 3)hf'(x) + (-\frac{3}{2} + 9 - 12 + \frac{9}{2})h^2f''(x)}{12} + \\ &+ \frac{(\frac{1}{2} + 3 - 8 + \frac{9}{2})h^3f'''(x) + (-\frac{1}{8} + \frac{3}{4} - 4 + \frac{27}{8})h^4f^{(4)}(x)}{12} = \\ &= \frac{12hf'(x) + O(h^5)}{12} = hf'(x) + O(h^5) \end{aligned} \tag{2}$$

Esercizio 2: Matlab utilizza la doppia precisione IEEE. Stabilire, pertanto, il nesso tra la variabile **eps** e la precisione di macchina di questa aritmetica.

Soluzione: Data la funzione $fl : I \rightarrow \mathcal{M}$, che associa ad ogni numero reale $x \in \mathcal{I}$, un corrispondente numero di macchina $fl(x) \in \mathcal{M}$, lo Standard in doppia precisione IEEE prevede la rappresentazione per arrotondamento del numero $fl(x)$.

Possiamo quindi affermare che se $x \in \mathcal{I}, x \neq 0$, allora:

$$fl(x) = x(1 + \epsilon_x), \quad |\epsilon_x| \leq u$$

In cui:

$$u = \frac{1}{2}b^{1-m} \quad \text{rappresentazione per arrotondamento}$$

Dove b è la base, m è il numero di cifre per la mantissa e ϵ_x l'errore relativo di rappresentazione.

Dato che nello standard IEEE in doppia precisione si utilizza la base $b = 2$ e un numero di cifre per la mantissa pari a $m = 53$, avremo che la precisione di macchina con rappresentazione per arrotondamento è data da: $u = 2^{-53} \approx 1.1102 * 10^{-16}$

In Matlab, la variabile **eps** contiene la precisione di macchina in base 10 che coincide con il valore u nel caso di rappresentazione per troncamento infatti vale: $eps = 2.2204 * 10^{-16}$.

Dato che **eps** rappresenta quindi, la distanza tra 1 e il successivo numero in virgola mobile, ovvero il valore: $x = 1 + u = 1 + 2^{-53}$, il quale è rappresentato da $fl(x) = 1$ dato che $u \leq eps$. L'errore relativo commesso su x perciò è:

$$|\epsilon_x| = \frac{|fl(x) - x|}{|x|} = \frac{|1 - (1 + 2^{-53})|}{|1 + 2^{-53}|} = \frac{2^{-53}}{1 + 2^{-53}} \leq 2^{-53} = u \leq eps$$

Esercizio 3: Spiegare il fenomeno della cancellazione numerica. Fare un esempio che la illustri, spiegandone i dettagli.

Soluzione: La cancellazione numerica si verifica quando si perdono delle cifre significative durante un'operazione di somma algebrica, con addendi quasi opposti. Questo è dovuto al fatto che la somma è un'operazione malcondizionata ed è possibile studiarlo, verificando il condizionamento di $y = x_1 + x_2$, $x_1, x_2 \in \mathbb{R}$ con $x_1 + x_2 \neq 0$.

Siano ϵ_1 e ϵ_2 gli errori relativi sui dati iniziali e considerando che non venga introdotto nessun nuovo errore nel calcolo della somma precedente, si ottiene:

$$y(1 + \epsilon_y) = x_1(1 + \epsilon_1) + x_2(1 + \epsilon_2)$$

Da cui si può ricavare che:

$$|\epsilon_y| \leq \frac{|x_1| + |x_2|}{|x_1 + x_2|} \epsilon_x \equiv k \epsilon_x \quad \text{con} \quad \epsilon_x = \max\{|\epsilon_1|, |\epsilon_2|\}$$

Il numero k , quindi, indica il numero di condizionamento, che può essere arbitrariamente grande, nel caso di due addendi quasi opposti tra loro. Ciò significa che l'operazione di somma tra numeri quasi opposti è malcondizionata.

Per esempio, supponiamo di voler calcolare $y = 0.2345666 - 0.2345111 \equiv 0.0000555$. Se utilizziamo una rappresentazione per arrotondamento alla quarta cifra significativa, otteniamo:

$$\tilde{y} = 2.346 * 10^{-1} - 2.345 * 10^{-1} = 1 * 10^{-4}$$

L'errore relativo che commettiamo su y sarà quindi:

$$|\epsilon_y| = \left| \frac{5.55 * 10^{-5} - 1 * 10^{-4}}{5.55 * 10^{-5}} \right| \simeq 0.8018$$

Andando quindi a calcolare il numero di condizionamento k , otteniamo:

$$k = \frac{|x_1| + |x_2|}{|x_1 - x_2|} = \frac{|0.2345666| + |-0.2345111|}{|0.2345666 - 0.2345111|} = 8.45 * 10^3$$

perciò avendo un numero notevolmente alto, la somma precedente è malcondizionata e abbiamo una perdita di cifre significative.

Esercizio 4: Scrivere una function Matlab, `radice(x)` che, avendo in ingresso un numero x non negativo, calcoli $\sqrt[6]{x}$ utilizzando solo operazioni algebriche elementari, con la massima precisione possibile. Confrontare con il risultato fornito da $x^{(1/6)}$ per 20 valori di x , equispaziati logaritmicamente nell'intervallo $[1e-10, 1e10]$, tabulando i risultati in modo che si possa verificare che si è ottenuta la massima precisione possibile.

Soluzione: E' possibile notare che la radice sesta di un numero k sia data dalla soluzione positiva dell'equazione $x^6 = k$, che può essere riscritta come $x^6 - k = 0$. Ciò equivale a ricercare la radice della funzione nel semiasse positivo delle ascisse della funzione $f(x) = x^6 - k$, è possibile quindi usare il metodo di Newton per trovarla. Quindi il passo iterativo dell'algoritmo diventa:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = x_n - \frac{x_n^6 - k}{6x_n^5} = x_n - \frac{1}{6}(x_n - \frac{k}{x_n^5})$$

Da questo risultato si giunge nel formulare il seguente metodo iterativo che converge verso la radice sesta di un numero x :

1. Si stima un valore iniziale di partenza x_0
2. Dopodiché si pone $x_{n+1} = x_n + \Delta x_n$ in cui $\Delta x_n = (\frac{x}{x_n^5} - x_n) \frac{1}{6} \quad \forall n = 0, 1, 2, \dots$
3. Viene reiterato il passo precedente finchè non risulta che la differenza $|x_{n+1} - x_n|$ sia minore di una precisione scelta

Traducendo il precedente algoritmo in Matlab, otteniamo il seguente codice:

```

1 function y = radice(x)
2 %
3 % y = radice(x)
4 %
5 % Calcola la radice sesta di x utilizzando solo operazioni algebriche
6 % elementari con la massima precisione possibile.
7 %
8 % Input: x, numero di cui calcolare la radice
9 % Output: y, risultato dell'operazione di radice
10 %
11 %
12
13 if x <= 0, error("Il radicando non deve essere negativo"); end
14 x0 = 1;
15 x1 = x0 + (x/(x0^5) - x0)/6;
16 err=1;
17 while err >= eps*(1+abs(x0))
18     x0 = x1;
```

```

19     x1 = x0 + (x/(x0*x0*x0*x0*x0)-x0)/6;
20     err=abs(x1-x0);
21 end
22 y=x1;
23 return
24 end

```

Con il seguente codice si vanno a generare 20 valori equispaziati logaritmicamente nell'intervallo $[1e-10, 1e10]$ e vengono comparate la rappresentazione esatta della radice con quella della funzione radice precedente:

```

1  approssimato = zeros(1,20);
2  esatto = zeros(1,20);
3  errore = zeros(1,20);
4  x = logspace(log10(1e-10),log10(1e10),20);
5  for i = 1:20
6      approssimato(i) = radice(x(i));
7      esatto(i) = (x(i))^(1/6);
8      errore(i) = abs(approssimato(i)-esatto(i));
9  end
10 variabili = {'n', 'approssimato', 'esatto', 'errore'};
11 table(x',approssimato',esatto', errore', 'VariableNames',variabili);

```

Si ottengono così i seguenti risultati:

	n	approssimato	esatto	errore
1	1.0000e-10	0.0215	0.0215	3.4694e-18
2	1.1288e-09	0.0323	0.0323	6.9389e-18
3	1.2743e-08	0.0483	0.0483	6.9389e-18
4	1.4384e-07	0.0724	0.0724	1.3878e-17
5	1.6238e-06	0.1084	0.1084	2.7756e-17
6	1.8330e-05	0.1624	0.1624	2.7756e-17
7	2.0691e-04	0.2432	0.2432	2.7756e-17
8	0.0023	0.3643	0.3643	0
9	0.0264	0.5456	0.5456	0
10	0.2976	0.8171	0.8171	0
11	3.3598	1.2238	1.2238	0
12	37.9269	1.8330	1.8330	2.2204e-16
13	428.1332	2.7453	2.7453	0
14	4.8329e+03	4.1118	4.1118	8.8818e-16
15	5.4556e+04	6.1585	6.1585	0
16	6.1585e+05	9.2239	9.2239	0
17	6.9519e+06	13.8150	13.8150	0
18	7.8476e+07	20.6914	20.6914	3.5527e-15
19	8.8587e+08	30.9905	30.9905	7.1054e-15
20	1.0000e+10	46.4159	46.4159	7.1054e-15

E' possibile notare come i valori ottenuti dalla funzione si avvicinino molto ai valori reali, con una tolleranza che è vicina ad **eps**.

Esercizio 5: Scrivere function Matlab distinte che implementino efficientemente i metodi di Newton e delle secanti per la ricerca degli zeri di una funzione $f(x)$. Per tutti i metodi, utilizzare come criterio di arresto:

$$|x_{n+1} - x_n| \leq tol \cdot (1 + |x_n|)$$

essendo tol una opportuna tolleranza specificata in ingresso. Curare particolarmente la robustezza del codice.

Soluzione:

CODICE Matlab per il metodo di Newton

```

1 function [x,nit] = newton(f,f1,x0,tolx,maxit)
2 % Il metodo di Newton serve per determinare una approssimazione della
3 % radice a partire da un'approssimazione iniziale.
4 %
5 % Input: f = funzione di cui vogliamo trovare la radice
6 %         f1 = derivata prima della funzione f
7 %         x0 = approssimazione iniziale della radice
8 %         tolx = tolleranza fissata
9 %         maxit = massimo numero di iterazioni fissato
10 %
11 % Output: x = radice della funzione f
12 %          nit = numero di iterazioni svolte, vale -1 se la tolleranza non
13 %              e' soddisfatta entro maxit o la derivata si annulla
14 %
15 if nargin<4, error('Argomenti in input non sufficienti');
16 elseif nargin==4, maxit=100;end
17 if tolx<eps, error('Tolleranza non valida');end
18
19 x=x0;
20 nit=-1;
21 for i=1:maxit
22     fx = feval(f,x);
23     f1x = feval(f1,x);
24     if f1x == 0, error('Derivata prima uguale a 0'); end
25     x = x - fx/f1x;
26     if abs(x-x0)<=tolx * (1+abs(x0))
27         nit=i;
28         break;
29     else
30         x0 = x;
31     end
32 end
33
34 if nit == -1, disp('Tolleranza desiderata non raggiunta');end
35 return
36 end

```

CODICE Matlab per il metodo delle secanti

```

1 function [x,nit] = secanti(f,x0,x1,tolx,maxit)
2 %
3 % Il metodo delle secanti serve per determinare una approssimazione della
4 % radice di f(x)=0 a partire da due valori iniziali x0 e x1.
5 %
6 % Input: f = funzione di cui vogliamo trovare la radice
7 %         x0 = approssimazione iniziale della radice
8 %         x1 = approssimazione iniziale della radice

```

```

9 %           tolx = tolleranza fissata
10 %           maxit = massimo numero di iterazioni fissato
11 %
12 % Output: x = radice della funzione f
13 %           nit = numero di iterazioni svolte, vale -1 se la tolleranza non
14 %           e' soddisfatta entro maxit o la derivata si annulla
15 %
16 if nargin<4, error('Argomenti in input non sufficienti')
17 elseif nargin==4, maxit=100;end
18 if tolx<eps, error('Tolleranza non valida');end
19 nit=-1;
20 for i=1:maxit
21     fx0 = feval(f,x0);
22     fx1 = feval(f,x1);
23     if fx1-fx0 == 0, error('Il denominatore e'' uguale a 0');end
24     x = (fx1*x0-fx0*x1)/(fx1-fx0);
25     x0 = x1;
26     x1 = x;
27     if abs(x-x0)<=tolx * (1+abs(x0))
28         nit=i;
29         break;
30     end
31 end
32
33 if nit == -1, disp('Tolleranza desiderata non raggiunta');end
34 return
35 end

```

Esercizio 6: Utilizzare le *function* del precedente esercizio per determinare una approssimazione della radice della funzione:

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

per $tol = 10^{-3}, 10^{-6}, 10^{-9}, 10^{-12}$, partendo da $x_0 = 0$ (e $x_1 = 0.1$ per il metodo delle secanti). Tabulare i risultati, in modo da confrontare il costo computazionale di ciascun metodo.

Soluzione:

tol	Radici Newton	iterazioni	Radici secanti	iterazioni
10^{-3}	7.390851333852840e-01	4	7.390985629062998e-01	4
10^{-6}	7.390851332151607e-01	5	7.390851332151466e-01	6
10^{-9}	7.390851332151607e-01	5	7.390851332151607e-01	7
10^{-12}	7.390851332151607e-01	6	7.390851332151607e-01	7

Considerando che per ciascuna iterazione del metodo di Newton si eseguono 2 valutazioni funzionali mentre per il metodo delle secanti ne viene fatta solo una, si ottiene che Newton utilizza un costo computazionale maggiore rispetto al metodo delle secanti in questo caso.

Esercizio 7: Utilizzare le *function* dell'Esercizio 5 per determinare una approssimazione della radice della funzione:

$$f(x) = [x - \cos(x)]^5$$

per $tol = 10^{-3}, 10^{-6}, 10^{-9}, 10^{-12}$, partendo da $x_0 = 0$ (e $x_1 = 0.1$ per il metodo delle secanti). Tabulare i risultati, in modo da confrontare il costo computazionale e l'accuratezza di ciascun metodo. Commentare i risultati ottenuti.

Soluzione:

tol	Radici Newton	iterazioni	Radici secanti	iterazioni
10^{-3}	0.732640697751109	20	0.730145017727562	26
10^{-6}	0.739078762321033	51	0.739075266476228	70
10^{-9}	0.739085126905744	82	0.739085038011533	-1
10^{-12}	0.7390851331015	-1	0.739085038011533	-1

Com'è possibile notare dai precedenti risultati, in entrambi i metodi, essendo la funzione una radice multipla, si eseguono molte più iterazioni rispetto alla funzione dell'Esercizio 6, in alcuni casi neanche convergono. In particolare risulta che il metodo di Newton non converga su tolleranza pari a $1e-12$ mentre il metodo delle secanti non converga nel caso la tolleranza sia pari a: $1e-9$, $1e-12$.

In conclusione, in caso di radici multiple, non porta alcun vantaggio utilizzare metodi diversi da quello di Newton, anzi avremo un aumento considerevole nel numero di iterazioni.

Esercizio 8: Scrivere una function Matlab,

`function x = mialu(A,b)`

che, data in ingresso una matrice A ed un vettore b , calcoli la soluzione del sistema lineare $Ax = b$ con il metodo di fattorizzazione LU con *pivoting parziale*. Curare particolarmente la scrittura e l'efficienza della function, e validarla su due esempi non banali, generati casualmente, di cui sia nota la soluzione.

Soluzione: CODICE Matlab per funzione mialu

```

1 function x = mialu(A,b)
2 % x = mialu(A,b)
3 % data in ingresso una matrice A ed un vettore b, calcoli la soluzione del
4 % sistema lineare Ax = b con il metodo di fattorizzazione LU con pivoting
5 % parziale.
6 % Input: A = matrice in ingresso che va fattorizzata LU con il metodo di
7 %           pivoting parziale
8 %           b = vettore dei termini noti
9 % Output: x = vettore soluzione di Ax=b
10 %
11
12 [m,n] = size(A);
13 dimb = length(b);
14 if m ~= n
15     error("La matrice dei coefficienti A deve essere quadrata")
16 end
17 if m ~= dimb
18     error("La matrice A ed il vettore b hanno dimensioni discordanti")
19 end
20 p = [1:n];
21 for i = 1:n-1
22     [mi,ki] = max(abs(A(i:n,i)));
23     if mi == 0
```



```

24         error('La matrice non puo'' essere singolare');
25     end
26     ki = ki + i - 1;
27     if ki > i
28         % inverto la riga i-esima e ki-esima
29         A([i,ki],:) = A([ki,i],:);
30         % stessa cosa nel vettore delle permutazioni
31         p([i,ki]) = p([ki,i]);
32     end
33     A(i+1:n,i) = A(i+1:n,i)/A(i,i);
34     A(i+1:n,i+1:n) = A(i+1:n,i+1:n) - A(i+1:n,i)*A(i,i+1:n);
35 end
36 x = b(p);
37 for i = 1:n
38     x(i+1:n) = x(i+1:n)-A(i+1:n,i)*x(i);
39 end
40 x(n) = x(n)/A(n,n);
41 for i = n-1:-1:1
42     x(1:i) = x(1:i) - A(1:i,i+1)*x(i+1);
43     x(i) = x(i)/A(i,i);
44 end
45 return
46 end

```

Esempi:

$$A = \begin{pmatrix} -3 & 5 & -4 \\ 3 & -3 & -4 \\ -1 & -3 & 4 \end{pmatrix} \quad x = \begin{pmatrix} 1 \\ 1 \\ -4 \end{pmatrix} \quad b = \begin{pmatrix} 18 \\ 16 \\ -20 \end{pmatrix} \quad (3)$$

$$A = \begin{pmatrix} 3 & -1 & 1 & 1 \\ 0 & 3 & -3 & 2 \\ 2 & 2 & 2 & 2 \\ -3 & 3 & 3 & -1 \end{pmatrix} \quad x = \begin{pmatrix} 1 \\ -2 \\ 1 \\ -3 \end{pmatrix} \quad b = \begin{pmatrix} 3 \\ -15 \\ -6 \\ -3 \end{pmatrix} \quad (4)$$

In entrambi i casi sono stati creati i valori della matrice A e del vettore x in modo casuale, mentre il vettore dei termini noti b , è stato calcolato tramite il prodotto $A \cdot x$. Una volta inseriti i valori di A e b nella funzione `mialu(A,b)`, questa ha restituito il vettore x correttamente.

Esercizio 9: Scrivere una function Matlab,

`function x = mialdl(A,b)`

che, data in ingresso una matrice A ed un vettore b , calcoli la soluzione del corrispondente sistema lineare utilizzando la fattorizzazione LDL^T . Curare particolarmente la scrittura e l'efficienza della function, e validarla su due esempi non banali, generati casualmente, di cui sia nota la soluzione.

Soluzione: CODICE Matlab per funzione `mialdl`

```

1 function x = mialdl(A,b)
2 % x = mialdl(A,b)
3 % data in ingresso una matrice A ed un vettore b, calcoli la soluzione del
4 % sistema lineare Ax = b con il metodo di fattorizzazione LDL^T.
5
6 % Input: A = matrice in ingresso che va fattorizzata LDL^T
7 %       b = vettore dei termini noti
8 % Output: x = vettore soluzione di Ax=b
9 %
10 [m,n] = size(A);
11 dimb = length(b);
12 if m ~= n
13     error("La matrice dei coefficienti A deve essere quadrata")
14 end
15 if m ~= dimb
16     error("La matrice A ed il vettore b hanno dimensioni discordanti")
17 end
18 if A(1,1) <= 0, error('la matrice non e'' sdp'); end
19 A(2:n,1) = A(2:n,1)/A(1,1);
20 for j = 2:n
21     v = (A(j,1:j-1).') .* diag(A(1:j-1,1:j-1));
22     A(j,j) = A(j,j) - A(j,1:j-1)*v;
23     if A(j,j) <= 0, error('la matrice non e'' sdp'); end
24     A(j+1:n,j) = (A(j+1:n,j) - A(j+1:n,1:j-1) * v)/A(j,j);
25 end
26 x = b;
27 for i=1:n
28     x(i+1:n) = x(i+1:n)-(A(i+1:n,i)*x(i));
29 end
30 x = x./diag(A);
31 for i=n:-1:2
32     x(1:i-1) = x(1:i-1)-A(i,1:i-1).'*x(i);
33 end
34 end

```

Esempi:

$$A = \begin{pmatrix} 5 & 3 & -1 \\ 3 & 6 & -1 \\ -1 & -1 & 6 \end{pmatrix} \quad x = \begin{pmatrix} -3 \\ 0 \\ 0 \end{pmatrix} \quad b = \begin{pmatrix} -15 \\ -9 \\ 3 \end{pmatrix} \quad (5)$$

$$A = \begin{pmatrix} 3 & -2 & 1 & 0 \\ -2 & 4 & 2 & 0 \\ 1 & 2 & 5 & -3 \\ 0 & 0 & -3 & 5 \end{pmatrix} \quad x = \begin{pmatrix} 1 \\ 1 \\ -2 \\ 1 \end{pmatrix} \quad b = \begin{pmatrix} -1 \\ -2 \\ -4 \\ 1 \end{pmatrix} \quad (6)$$

In entrambi i casi sono stati creati i valori della matrice A e del vettore x in modo casuale, mentre il vettore dei termini noti b , è stato calcolato tramite il prodotto $A \cdot x$. Una volta inseriti i valori di A e b nella funzione `mialdl(A,b)`, questa ha restituito il vettore x correttamente.

Esercizio 10: Scrivere una function Matlab,

`function [x,nr] = miaqr(A,b)`

che, data in ingresso la matrice A $m \times n$, con $m \geq n = \text{rank}(A)$, ed un vettore b di lunghezza m , calcoli la soluzione del sistema lineare $Ax = b$ nel senso dei minimi quadrati e, inoltre, la norma, nr , del corrispondente vettore residuo. Curare particolarmente la scrittura e l'efficienza della function. Validare la function `miaqr` su due esempi non banali, generati casualmente, confrontando la soluzione ottenuta con quella calcolata con l'operatore Matlab \

Soluzione: CODICE Matlab per funzione `mialdl`

```

1 function [x,nr] = miaqr(A,b)
2 %
3 % [x,nr] = miaqr(A,b)
4 %
5 % La funzione miaqr fattorizza QR la matrice in ingresso A, dopodiche'
6 % restituisce la soluzione x del sistema Ax=b insieme alla norma del
7 % vettore residuo
8 %
9 % Input: A = matrice in ingresso
10 %         b = vettore dei termini noti
11 % Output: x = soluzione del sistema
12 %          nr = norma del vettore residuo
13 %
14
15 [m,n] = size(A);
16 dimb = length(b);
17 if n > m, error('Dimensioni matrice A errate'); end
18 if dimb ~= m, error('Dimensione vettore dei termini noti sbagliata'); end
19 for i=1:n
20     alfa = norm(A(i:m,i));
21     if alfa == 0, error('La matrice non ha rango massimo'), end
22     if A(i,i) >= 0, alfa = -alfa; end
23     v1 = A(i,i) - alfa;
24     A(i,i) = alfa;
25     A(i+1:m,i) = A(i+1:m,i)/v1;
26     beta = -v1/alfa;
27     A(i:m,i+1:n) = A(i:m,i+1:n) - (beta*[1; A(i+1:m,i)])*...
28         ([1; A(i+1:m,i)]' * A(i:m,i+1:n));
29 end
30 for i=1:n
31     v = [1; A(i+1:m,i)];
32     beta = 2/(v'*v);
33     b(i:dimb) = b(i:dimb) - (beta*(v'*b(i:dimb)))*v;
34 end
35 for i=n:-1:1
36     b(i) = b(i)/A(i,i);
37     b(1:i-1) = b(1:i-1) - A(1:i-1,i)*b(i);
38 end
39 x = b(1:n);
40 nr = norm(b(n+1:m));
41 end

```

Esempio 1: Data la matrice A di dimensioni 4×3 e il vettore b (entrambi generati casualmente):

$$A = \begin{pmatrix} 4 & -5 & 2 \\ 3 & -5 & -4 \\ 3 & 3 & 2 \\ -5 & 5 & -4 \end{pmatrix} \quad b = \begin{pmatrix} -8 \\ 3 \\ -4 \\ 3 \end{pmatrix} \quad (7)$$

La soluzione generata dalla `function miaqr` è:

$$x_{Miaqr} = \begin{pmatrix} -5.4619e-01 \\ -4.1315e-02 \\ -9.0373e-01 \end{pmatrix}, \text{ con norma } nr = 5.3356e-00$$

Mentre la soluzione generata da $x = A \setminus b$ è:

$$x = \begin{pmatrix} -5.4619e-01 \\ -4.1315e-02 \\ -9.0373e-01 \end{pmatrix}$$

Esempio 2: Data la matrice A di dimensioni 5×4 e il vettore b (entrambi generati casualmente):

$$A = \begin{pmatrix} 3 & 2 & -2 & -4 \\ -3 & 0 & -3 & -4 \\ -1 & -4 & 0 & 3 \\ 1 & 3 & -4 & -2 \\ -3 & -4 & -1 & 1 \end{pmatrix} \quad b = \begin{pmatrix} 10 \\ -1 \\ 4 \\ 5 \\ -1 \end{pmatrix} \quad (8)$$

La soluzione generata dalla `function miaqr` è:

$$x_{Miaqr} = \begin{pmatrix} 2.8454e+00 \\ -1.8283e+00 \\ -1.7138e+00 \\ -4.5249e-01 \end{pmatrix}, \text{ con norma } nr = 1.4966e+00$$

Mentre la soluzione generata da $x = A \setminus b$ è:

$$x = \begin{pmatrix} 2.8454e+00 \\ -1.8283e+00 \\ -1.7138e+00 \\ -4.5249e-01 \end{pmatrix}$$

Esercizio 11: Data la function Matlab

```

1 function [A1,A2,b1,b2] = linsis(n,simme)
2 %
3 %
4 rng(0);
5 [q1,r1] = qr(rand(n));
6 if nargin==2
7     q2 = q1';
8 else
9     [q2,r1] = qr(rand(n));
10 end
11 A1 = q1*diag([1 2/n:1/n:1])*q2;
12 A2 = q1*diag([1e-10 2/n:1/n:1])*q2;
13 b1 = sum(A1,2);
14 b2 = sum(A2,2);
15 return

```

che crea sistemi lineari casuali di dimensione n con soluzione nota,

$$A_1 x = b_1, \quad A_2 x = b_2, \quad x = (1, \dots, 1)^T \in \mathbb{R}^n,$$

risolvere, utilizzando la `function mialu`, i sistemi lineari generati da `[A1,A2,b1,b2]=linsis(5)`. Commentare l'accuratezza dei risultati ottenuti, dandone spiegazione esaustiva.

Soluzione: Innanzitutto bisogna notare che eseguendo la function `linsis(5)`, che ha in input un solo argomento, andiamo a creare matrici non simmetriche. Perciò utilizzando i risultati come input della function `mialu`, otteniamo i seguenti risultati:

1. Nel primo caso, ovvero calcolando $\mathbf{x}_1 = \text{mialu}(\mathbf{A}_1, \mathbf{b}_1)$, il vettore risultante è:

$$x_1 = \begin{pmatrix} 9.999999999999999e-01 \\ 9.999999999999998e-01 \\ 9.999999999999998e-01 \\ 1 \\ 9.999999999999998e-01 \end{pmatrix}$$

Il risultato è pressoché identico al risultato atteso. D'altra parte il numero di condizionamento K della matrice \mathbf{A}_1 , dato dalla funzione `cond(A1)` è pari a $K = 2.5000e+00$. Ciò significa che il problema è quindi ben condizionato, essendo K piccolo.

2. Nel secondo caso invece, ovvero calcolando $\mathbf{x}_2 = \text{mialu}(\mathbf{A}_2, \mathbf{b}_2)$, il vettore risultante è:

$$x_2 = \begin{pmatrix} 9.999996476574766e-01 \\ 1.000000446226050e+00 \\ 1.000000098875194e+00 \\ 1.000000207059384e+00 \\ 1.000000011600807e+00 \end{pmatrix}$$

Il cui numero di condizionamento della matrice \mathbf{A}_2 , dato dalla funzione `cond(A2)` è pari a $K = 9.999995892902628e+09$. Dato che il numero ottenuto è molto grande, significa che il problema risulta mal condizionato.

Questo è dovuto all'operazione con cui otteniamo la matrice \mathbf{A}_2 , in particolare all'operazione: `diag([1e-10 2/n:1/n:1])`, in cui il primo elemento della matrice diagonale è molto vicino allo zero:

$$D = \begin{pmatrix} 1e-10 & 0 & 0 & 0 & 0 \\ 0 & 0.4 & 0 & 0 & 0 \\ 0 & 0 & 0.6 & 0 & 0 \\ 0 & 0 & 0 & 0.8 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Infatti, ricordando la costruzione della matrice \mathbf{A}_2 mediante $A = Q_1 * D * Q_2$, per calcolare la matrice inversa basterà ricordare che Q_1 e Q_2 sono matrici ortogonali, ottenendo: $A^{-1} = Q_2^{-1} * D^{-1} * Q_1^{-1} = Q_2^T * D^{-1} * Q_1^T$.

Inoltre si può notare che la matrice inversa di una matrice diagonale è una matrice diagonale dove gli elementi diagonali sono il reciproco rispetto alla matrice di partenza:

$$D^{-1} = \begin{pmatrix} 1e10 & 0 & 0 & 0 & 0 \\ 0 & 2.5 & 0 & 0 & 0 \\ 0 & 0 & 1.667 & 0 & 0 \\ 0 & 0 & 0 & 1.25 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Calcolando quindi la matrice inversa con la formula sopra illustrata si ottiene che moltiplicando Q_1^T per D si realizza una matrice dove la prima colonna avrà elementi nell'ordine di $1e9$, tale matrice moltiplicata per Q_2^T infine produce una matrice dove tutti gli elementi raggiungono valori molto elevati, come si può notare dalla norma della matrice inversa:

$$\|A_1\|_2 = 1 \quad \|A_2\|_2 = 1 \quad \|A_1^{-1}\|_2 = 2.5 \quad \|A_2^{-1}\|_2 = 1.0000e+10$$

Da questi risultati si evince quindi che la costruzione di \mathbf{A}_2 risulta peggiore rispetto a quella di \mathbf{A}_1 visto che risulta più sensibile ad errori.

Esercizio 12: Risolvere, utilizzando la *function* `mialdlt`, i sistemi lineari generati da `[A1,A2,b1,b2]=linsis(5,1)`. Commentare l'accuratezza dei risultati ottenuti, dandone spiegazione esaustiva.

Soluzione: Similmente al caso precedente usando la *function* `mialdlt`, otteniamo i seguenti risultati:

1. Nel primo caso, ovvero calcolando `x1 = mialdlt(A1,b1)`, il vettore risultante è:

$$x_1 = \begin{pmatrix} 1 \\ 9.999999999999994e-01 \\ 9.999999999999999e-01 \\ 1.000000000000000e+00 \\ 1.000000000000000e+00 \end{pmatrix}$$

Il risultato è pressoché identico al risultato atteso. D'altra parte il numero di condizionamento K della matrice A_1 , dato dalla funzione `cond(A1)` è pari a $K = 2.5000e+00$, quindi il problema è ben condizionato.

2. Nel secondo caso invece, ovvero calcolando `x2 = mialdlt(A2,b2)`, il vettore risultante è:

$$x_2 = \begin{pmatrix} 1.000000052293492e+00 \\ 1.000000058138759e+00 \\ 1.000000008150719e+00 \\ 1.000000058625536e+00 \\ 1.000000040588329e+00 \end{pmatrix}$$

Il cui numero di condizionamento della matrice A_2 , dato dalla funzione `cond(A2)` è pari a $K = 9.999995645805687e+09$. Dato che il numero ottenuto è molto grande significa che il problema è mal condizionato.

Tuttavia a differenza del caso precedente vengono costruite matrici simmetriche definite positive infatti:

- costruendo la matrice con $Q_1 * D * Q_1^T$ si ha che la transposta è data da: $(Q_1 * D * Q_1^T)^T = Q_1 * D^T * Q_1^T = Q_1 * D * Q_1^T$.
- $\forall x \in R^n, xQ_1 * D * Q_1^T x^T = (xQ_1) * D * (xQ_1)^T$, sostituendo $xQ_1 = y \in R^n$ con $y \neq 0$ (perché la matrice Q_1 è non singolare), avremo che $y * D * y^T > 0$ perché gli elementi della matrice diagonale sono tutti positivi, e quindi la matrice è definita positiva

Questa costruzione ha anche un'altra particolarità, ovvero le matrici A_1 e A_2 sono simili alla matrice D . Infatti per definizione, la matrice A e B sono simili se esiste una matrice P invertibile tale che: $A = P^{-1} * B * P$. Considerando $P = Q_1^T$ e $B = D$ allora si ha che: $A = (Q_1^T)^{-1} * D * Q_1^T = (Q_1^{-1})^T * D * Q_1^T = (Q_1^T)^T * D * Q_1^T = Q_1 * D * Q_1^T$. Quindi ciò comporta che le matrici A_1 e A_2 hanno gli stessi autovalori, e quindi lo stesso raggio spettrale della matrice diagonale. Anche la matrice inversa di A_1 e A_2 risulta essere simile, come si può notare anche da $A^{-1} = (Q_1^T) * D^{-1} * Q_1$. Quindi in generale avremo che:

$$\begin{aligned} \|A_1\| &\geq \rho(D_1) = 1 \\ \|A_1^{-1}\| &\geq \rho(D_1^{-1}) = 2.5 \\ \|A_2\| &\geq \rho(D_2) = 1 \\ \|A_2^{-1}\| &\geq \rho(D_2^{-1}) = 1e10 \end{aligned}$$

Per cui data una norma qualsiasi, abbiamo che il numero di condizionamento della prima matrice sarà $\|A_1\| * \|A_1^{-1}\| \geq 2.5$, mentre per la seconda $\|A_2\| * \|A_2^{-1}\| \geq 1e10$. Ciò tuttavia è vero solo in condizioni ottimali, infatti entrambe le matrici sono soggette ad errori di round-off dovuti alla loro costruzione. Ciò si può notare anche solo usando il comando `isSymmetric()` che per entrambe le matrici risulta essere falso, si può anche notare come il numero di condizionamento della seconda matrice calcolato tramite Matlab va in contraddizione con quanto stabilito. Quindi il mal condizionamento del secondo sistema rispetto al primo, causa errori di imprecisione maggiori.

Esercizio 13: Utilizzare la *function* `miaqr` per risolvere, nel senso dei minimi quadrati, i sistemi lineari sovradeterminati

$$Ax = b, \quad (D*A)x = (D*b), \quad (D1*A)x = (D1*b),$$

definiti dai seguenti dati:

$$A = [1 \ 3 \ 2; 3 \ 5 \ 4; 5 \ 7 \ 6; 3 \ 6 \ 4; 1 \ 4 \ 2];$$

$$b = [15 \ 28 \ 41 \ 33 \ 22]';$$

$$D = \text{diag}(1:5);$$

$$D1 = \text{diag}(\pi * [1 \ 1 \ 1 \ 1 \ 1]).$$

Calcolare le corrispondenti soluzioni e residui, e commentare i risultati ottenuti.

Soluzione: Ricordando che risolvere il sistema nel senso dei minimi quadrati consiste in un problema di ottimizzazione nella forma:

$$\min_{x \in \mathbb{R}^n} \|Ax - b\|_2^2$$

dove per residuo si intende un vettore $r = Ax - b$, si nota che la norma euclidea al quadrato dei residui non è altro che il valore all'ottimo. Applicando quindi la *function* `miaqr` ai vari sistemi, si ottengono quindi le seguenti soluzioni, residui e norme euclidee al quadrato:

$$x_1 = \begin{pmatrix} 3.0000000000000008e+00 \\ 5.8000000000000001e+00 \\ -2.5000000000000008e+00 \end{pmatrix}; r_1 = \begin{pmatrix} 0.3999999999999993 \\ -3.5527136788005e-15 \\ -0.3999999999999999 \\ 0.7999999999999997 \\ -0.8000000000000004 \end{pmatrix}; \|r_1\|_2^2 = 1.6$$

$$x_2 = \begin{pmatrix} -6.025699862322442e-01 \\ 4.701698026617703e+00 \\ 1.758375401560388e+00 \end{pmatrix}; r_2 = \begin{pmatrix} 2.01927489674164 \\ 1.46856356126666 \\ -1.65213400642492 \\ 1.74391922900415 \\ -1.3951353832033 \end{pmatrix}; \|r_2\|_2^2 = 13.9513538320331$$

$$x_3 = \begin{pmatrix} 3.0000000000000001 \\ 5.8000000000000001 \\ -2.5000000000000003 \end{pmatrix}; r_3 = \begin{pmatrix} 1.25663706143589 \\ -5.6843418860808e-14 \\ -1.25663706143598 \\ 2.51327412287179 \\ -2.51327412287183 \end{pmatrix}; \|r_3\|_2^2 = 15.7913670417428$$

Dai risultati ottenuti è possibile notare una grande discrepanza tra quelli ottenuti con il primo e il secondo sistema, nonostante che risolvere il sistema lineare $Ax=b$ sia equivalente a risolvere il sistema lineare $(D*A)x = (D*b)$. Tuttavia possiamo notare che nel caso dei minimi quadrati ciò non è sempre vero. Infatti il problema di minimo nel secondo caso risulta essere:

$$\min_{x \in \mathbb{R}^n} \|(D * A)x - (D * b)\|_2^2 = \min_{x \in \mathbb{R}^n} \|D * (Ax - b)\|_2^2$$

Non è detto infatti che la soluzione x_1 risolva questo nuovo problema, anzi, provando a sostituire x_1 nel secondo sistema si ottiene come risultato che la funzione obiettivo ha come valore 27.8400000000001 che risulta essere una soluzione ben peggiore rispetto a quella data da x_2 .

Nel caso invece del terzo sistema, possiamo notare che la matrice altro non è che il risultato del prodotto $\pi * I$. Il problema in questo caso risulterà essere:

$$\min_{x \in \mathbb{R}^n} \|(\pi * A)x - (\pi * b)\|_2^2 = \min_{x \in \mathbb{R}^n} \|\pi * (Ax - b)\|_2^2 = \pi^2 * \min_{x \in \mathbb{R}^n} \|Ax - b\|_2^2$$

E' possibile notare che tale problema equivale a quello del primo sistema moltiplicato per uno scalare, per cui la soluzione di questo problema di minimo coincide con la soluzione di quella del primo sistema. Infatti le due soluzioni ad eccezione di qualche decimale coincidono, come si può inoltre notare dalla norma residua calcolata ponendo $x = x_1$ che risulta essere: 15.7913670417428, le cifre inesatte sono dovute da piccole imprecisioni delle operazioni in aritmetica finita, specialmente considerando l'approssimazione necessaria di π , non essendo possibile rappresentare il numero interamente in codice macchina.

Esercizio 14: Scrivere una function Matlab,
`[x,nit] = newton(fun,jacobian,x0,tol,maxit)`
 che implementi efficientemente il metodo di Newton per risolvere sistemi di equazioni non-lineari. Curare particolarmente il criterio di arresto, che deve essere analogo a quello usato nel caso scalare. La seconda variabile, se specificata, ritorna il numero di iterazioni eseguite. Prevedere opportuni valori di *default* per gli ultimi due parametri di ingresso.

Soluzione: Funzione che calcola il metodo di Newton per risolvere sistemi di equazioni nonlineari

```

1 function [x,nit] = newton(fun,jacobian,x0,tol,maxit)
2 % [x,nit] = newton(fun,jacobian,x0,tol,maxit)
3 % Funzione che implementa il metodo di Newton per sistemi
4 % di equazioni nonlineari
5 %
6 % Input:
7 % fun: function funzione di un sistema nonlineare
8 % jacobian: matrice jacobiana di fun
9 % x0: approssimazione iniziale
10 % tol: tolleranza richiesta
11 % maxit: massimo numero di iterazioni richiesto
12 % Output:
13 % x: risultato del metodo di Newton
14 % nit: numero di iterazioni effettuate
15 %
16 if nargin < 3
17     error('numero argomenti insufficienti');
18 elseif nargin==3
19     tol=1e-13;
20     maxit=10e3;
21 elseif nargin==4
22     maxit=10e3;
23 end
24 x=x0;
25 nit=-1;
26 if tol<0
27     error('la tolleranza non puo essere negativa');
28 end
29 if maxit<=0
30     error('maxit deve essere maggiore di 0');
31 end
32 [n,m] = size(jacobian);
33 nf = size(fun);
34 if (n~=m)
35     error('La matrice Jacobiana deve essere quadrata');
36 end
37 if n~=nf
38     error('Il numero di righe del vettore fun e della Jacobiana sono diversi');
39 end
40 for i=1:maxit
41     fx = feval(fun,x);
42     f1x = feval(jacobian,x);
43     dx=-f1x\(fx);
44     x = x + dx;
45     if norm(dx./(1+abs(x)),Inf)<=tol
46         nit=i;
47         break;
48     end
49 end
50

```

```
51 if nit == -1, disp('Il metodo di Newton per il sistema lineare non converge');end
52 return
53 end
```

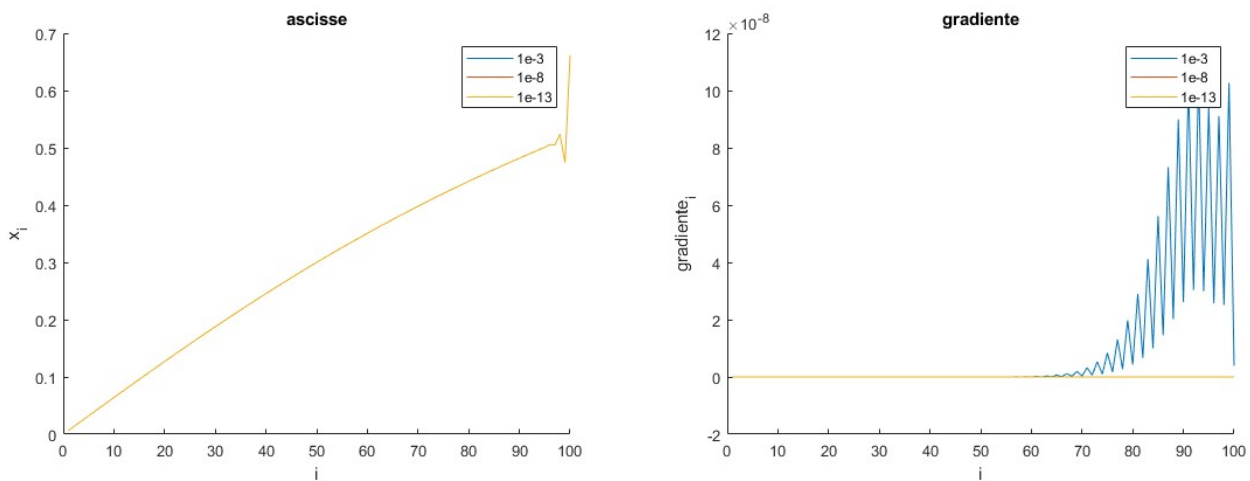
Esercizio 15: Usare la function del precedente esercizio per risolvere, a partire dal vettore iniziale nullo, il sistema nonlineare derivante dalla determinazione del punto stazionario della funzione:

$$f(\mathbf{x}) = \frac{1}{2} \mathbf{x}^T Q \mathbf{x} - \mathbf{e}^T \left[\sin\left(\frac{\pi}{2} \mathbf{x}\right) + \mathbf{x} \right], \quad \mathbf{e} = \frac{1}{100} \begin{pmatrix} 1 \\ 2 \\ \vdots \\ 100 \end{pmatrix} \in \mathbb{R}^{100},$$

$$Q = \begin{pmatrix} 2 & 1 & & \\ 1 & \ddots & \ddots & \\ & \ddots & \ddots & 1 \\ & & 1 & 2 \end{pmatrix} \in \mathbb{R}^{100 \times 100}, \quad \sin\left(\frac{\pi}{2} \mathbf{x}\right) = \begin{pmatrix} \sin\left(\frac{\pi}{2} x_1\right) \\ \sin\left(\frac{\pi}{2} x_2\right) \\ \vdots \\ \sin\left(\frac{\pi}{2} x_{100}\right) \end{pmatrix}$$

utilizzando tolleranze `tol = 1e-3, 1e-8, 1e-13`. Graficare la soluzione e tabulare in modo conveniente i risultati ottenuti.

Soluzione: Usando la funzione `newton` su $\nabla f(x)$ (definito in Matlab come `Q*x-((pi/2).*cos((pi/2)*x)+1).*e`) si ottiene:



Come si può notare dai grafici usando le tolleranze $1e-8$, $1e-13$ si ottiene che il gradiente è quasi nullo, ovvero le ascisse x_i risultano essere un punto stazionario. Tabulando infatti la norma 2 dei gradienti con le tolleranze si ha:

tol	norma del gradiente
1e-2	2.70412063725952e-07
1e-08	1.55513291374585e-15
1e-13	1.55513291374585e-15

Esercizio 16: Costruire una function, `lagrange.m`, avente la stessa sintassi della function `spline` di Matlab, che implementi, in modo vettoriale, la forma di Lagrange del polinomio interpolante una funzione.

Soluzione: CODICE Matlab per funzione Lagrange

```

1 function p = lagrange(x, y, xq)
2 %
3 %   p=lagrange(x, y, xq)
4 %
5 %   Calcolo del polinomio interpolante in base di Lagrange:
6 %
7 %   Input:
8 %   x: ascisse di interpolazione
9 %   y: valori della funzione interpolanda nelle ascisse di interpolazione
10 %   xq: valori su cui calcolare il valore del polinomio interpolante
11 %
12 %   Output:
13 %   p: vettore con i valori calcolati
14 %
15 n=length(x);
16 if length(unique(x))~=n
17     error("Le ascisse non sono distinte tra loro");
18 end
19 if length(y)~=n
20     error("I vettori x ed y devono avere la stessa lunghezza");
21 end
22 p=zeros(size(xq));
23 for i=1:n
24     p=p+y(i)*Lin(i,x,xq);
25 end
26 return
27 end

```

CODICE Matlab per funzione Lin

```

1 function L = Lin(i, xi, x)
2 %
3 %   L=Lin(i,xi,x)
4 %
5 %   Calcolo della base di Lagrange
6 %
7 %   Input:
8 %   i: indice
9 %   xi: ascisse di interpolazione
10 %   x: valori su cui calcolare la base di Lagrange
11 %
12 %   Output:
13 %   L: vettore con i valori della base di Lagrange con indice i
14 %
15 L=ones(size(x));
16 zi=xi(i);
17 xi(i)=[];
18 n=length(xi);
19 for j=1:n
20     L=L.*(x-xi(j));
21 end
22 L=L/prod(zi-xi);
23 return
24 end

```

Esercizio 17: Costruire una function, `newton.m`, avente la stessa sintassi della function `spline` di Matlab, che implementi, in modo vettoriale, la forma di Newton del polinomio interpolante una funzione.

Soluzione: CODICE Matlab per funzione Newton

```

1 function p = newton(x, y, xq)
2 %
3 %   p=newton(x, y, xq)
4 %
5 %   Calcolo del polinomio interpolante in base di Newton:
6 %
7 %   Input:
8 %   x: ascisse di interpolazione
9 %   y: valori della funzione interpolanda nelle ascisse di interpolazione
10 %   xq: valori su cui calcolare il valore del polinomio interpolante
11 %
12 %   Output:
13 %   p: vettore con i valori calcolati
14 %
15 n = length(x)-1;
16 if length(unique(x))~=n+1
17     error("Le ascisse non sono distinte tra loro");
18 end
19 if length(y)~=n+1
20     error("I vettori x ed y devono avere la stessa lunghezza");
21 end
22 for j=1:n
23     for i=n+1:-1:j+1
24         y(i)=(y(i)-y(i-1))/(x(i)-x(i-j));
25     end
26 end
27 p=horner(x,y,xq);
28 return
29 end

```

CODICE Matlab per l'algoritmo di Horner

```

1 function p = horner(x, f, xq)
2 %
3 %   p=horner(x, f, ,xq)
4 %
5 %   Algoritmo di horner generalizzato per il calcolo di un polinomio:
6 %
7 %   Input:
8 %   x: ascisse di interpolazione
9 %   f: valori delle differenze divise
10 %   xq: valori su cui calcolare il polinomio
11 %
12 %   Output:
13 %   p: vettore con i valori calcolati
14 %
15 n=length(x)-1;
16 p=ones(size(xq))*f(n+1);
17 for i=n:-1:1
18     p=p.*(xq-x(i))+f(i);
19 end
20 return
21 end

```

Esercizio 18: Costruire una function, `hermite.m`, avente sintassi

`yy = hermite(xi, fi, f1i, xx)`

che implementi, in modo vettoriale, il polinomio interpolante di Hermite.

Soluzione: CODICE Matlab per `hermite`

```

1 function yy = hermite(xi, fi, f1i, xx)
2 %
3 %   p=hermite(xi, fi, ,f1i, xx)
4 %
5 %   Calcolo del polinomio interpolante di Hermite:
6 %
7 %   Input:
8 %   xi: ascisse di interpolazione
9 %   fi: valori della funzione interpolanda nelle ascisse di interpolazione
10 %   f1i: valori della derivata prima della funzione interpolanda nelle
11 %   ascisse di interpolazione
12 %   xx: valori su cui calcolare il valore del polinomio interpolante
13 %
14 %   Output:
15 %   yy: vettore con i valori calcolati
16 %
17 n=length(xi)-1;
18 if length(unique(xi))~=n+1
19     error("Le ascisse non sono distinte tra loro");
20 end
21 if length(fi)~=n+1 || length(f1i)~=n+1
22     error("I vettori xi, fi e f1i devono avere la stessa lunghezza");
23 end
24 xi=repelem(xi,2);
25 fi=repelem(fi,2);
26 fi(2:2:end)=f1i;
27 for i = (2*n+1):-2:3
28     fi(i)=(fi(i)-fi(i-2))/(xi(i)-xi(i-1));
29 end
30 for j=2:2*n+1
31     for i = (2*n+2):-1:j+1
32         fi(i)= (fi(i)-fi(i-1))/(xi(i)-xi(i-j));
33     end
34 end
35 yy=horner(xi,fi,xx);
36 return
37 end

```

CODICE Matlab per l'algoritmo di horner

```

1 function p = horner(x, f, xq)
2 %
3 %   p=horner(x, f, ,xq)
4 %
5 %   Algoritmo di horner generalizzato per il calcolo di un polinomio:
6 %
7 %   Input:
8 %   x: ascisse di interpolazione
9 %   f: valori delle differenze divise
10 %   xq: valori su cui calcolare il polinomio
11 %
12 %   Output:
13 %   p: vettore con i valori calcolati

```

```

14 %
15 n=length(x)-1;
16 p=ones(size(xq))*f(n+1);
17 for i=n:-1:1
18     p=p.*(xq-x(i))+f(i);
19 end
20 return
21 end

```

Esercizio 19: Costruire una function Matlab che, specificato in ingresso il grado n del polinomio interpolante, e gli estremi dell'intervallo $[a, b]$, calcoli le corrispondenti ascisse di Chebyshev.

Soluzione: Calcolo delle ascisse di Chebyshev

```

1 function x = cheby(n,a,b)
2 %
3 %   x=cheby(n,a,b)
4 %
5 %   Calcolo delle ascisse di Chebyshev:
6 %
7 %   Input:
8 %   n: grado del polinomio
9 %   a,b: estremi dell'intervallo
10 %
11 %   Output:
12 %   x: vettore contenente le ascisse di Chebyshev
13 if n<=0 || n~=fix(n)
14     error('n deve essere un numero naturale');
15 elseif a>=b
16     error('l''intervallo non e'' corretto');
17 end
18 x=(a+b)/2 + cos((2*(n:-1:0)+1)*pi./(2*(n+1))) * (b-a)/2;
19 return
20 end

```

Esercizio 20: Costruire una function Matlab, con sintassi

`ll = lebesgue(a, b, nn, type)`

che approssimi la costante di Lebesgue per l'interpolazione polinomiale sull'intervallo $[a,b]$, per i polinomi di grado specificato nel vettore `nn`, utilizzando ascisse equidistanti, se `type=0`, o di Chebyshev, se `type=1` (utilizzare 10001 punti equispaziati nell'intervallo $[a,b]$ per ottenere ciascuna componente di `ll`). Graficare i risultati ottenuti, per `nn=1:100`, utilizzando $[a,b]=[0,1]$ e $[a,b]=[-3,7]$. Giustificare i risultati ottenuti.

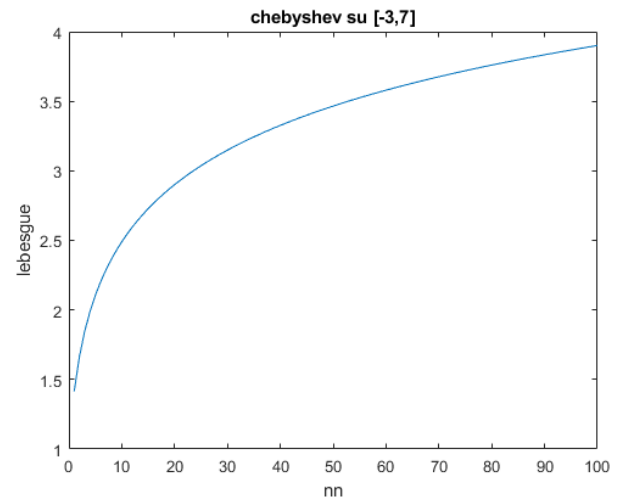
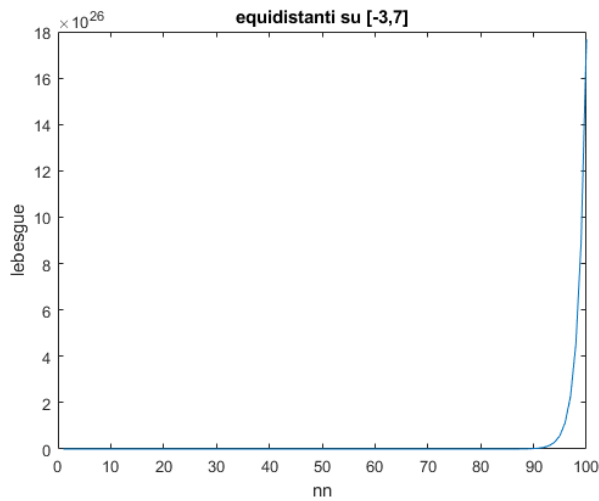
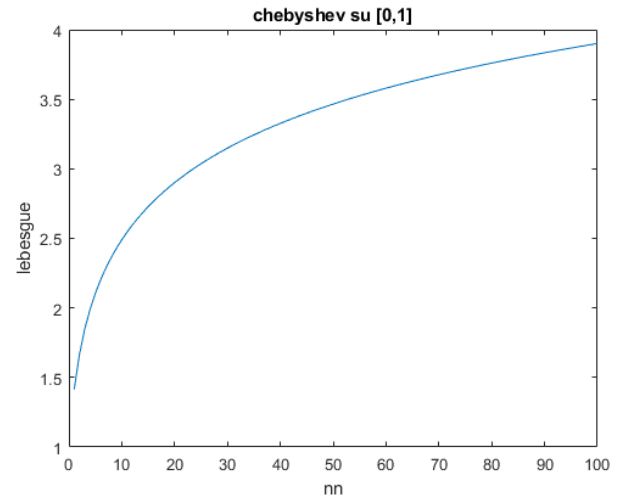
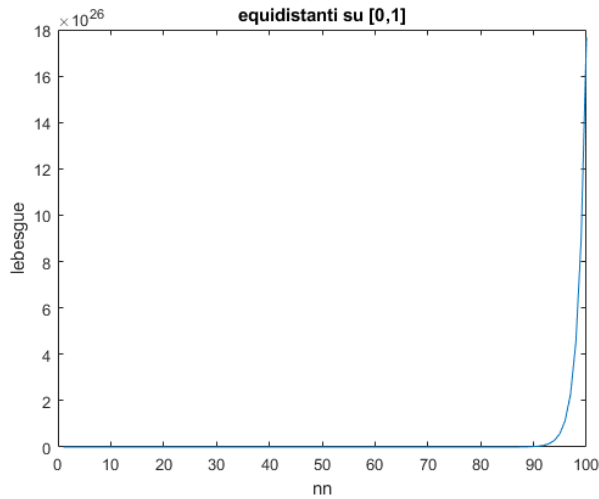
Soluzione: Calcolo della costate di Lebesgue:

```

1 function ll = lebesgue(a,b,nn,type)
2 %
3 %   ll = lebesgue(a,b,nn,type)
4 %   Calcola un'approssimazione della costante di Lebesgue per
5 %   l'interpolazione polinomiale sull'intervallo [a,b]
6 %
7 %   Input:
8 %   a,b: estremi intervallo
9 %   nn: grado dei polinomi
10 %   type:
11 %       0 - ascisse equidistanti
12 %       1 - ascisse di Chebyshev
13 %   Output:
14 %   ll: approssimazione costante di Lebesgue ottenuta
15 %
16 if nargin<4
17     error("inserire tutti i dati");
18 end
19 if a>b
20     error('a deve essere piu piccolo di b');
21 end
22 n=length(nn);
23 ll=zeros(1,length(nn));
24 f=0;
25 if type==1
26     f=@(n) cheby(n,a,b);
27 elseif type==0
28     f=@(n) linspace(a,b,n+1);
29 end
30 for i=1:n
31     xi=f(nn(i));
32     x=linspace(a,b,10001);
33     L=zeros(size(x));
34     for j=1:length(xi)
35         L=L+abs(Lin(j,xi,x));
36     end
37     ll(i)=max(L);
38 end
39 return
40 end

```

Graficando i risultati della funzione eseguita sugli intervalli $[0,1]$ e $[-3,7]$ con `type=0` e `type=1` si ottiene:



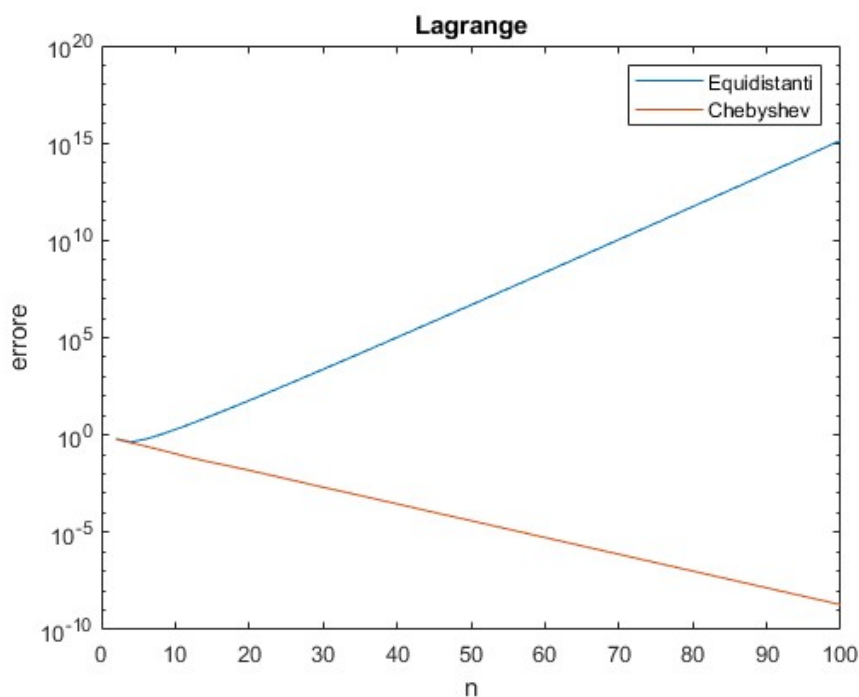
Si può vedere come la costante di Lebesgue cresca molto velocemente quando vengono usate ascisse equidistanti rispetto alle ascisse di Chebyshev. D'altronde Λ cresce almeno come $O(\log x)$, infatti usando ascisse equidistanti si ottiene una crescita esponenziale rispetto a quella ottimale ottenuta mediante le ascisse di Chebyshev $\Lambda \approx O(\log x)$, come si può notare dal grafico.

Esercizio 21: Utilizzando le function dei precedenti esercizi, graficare (in semilogy) l'andamento errore di interpolazione (utilizzare 10001 punti equispaziati nell'intervallo per ottenerne la stima) per la funzione di Runge,

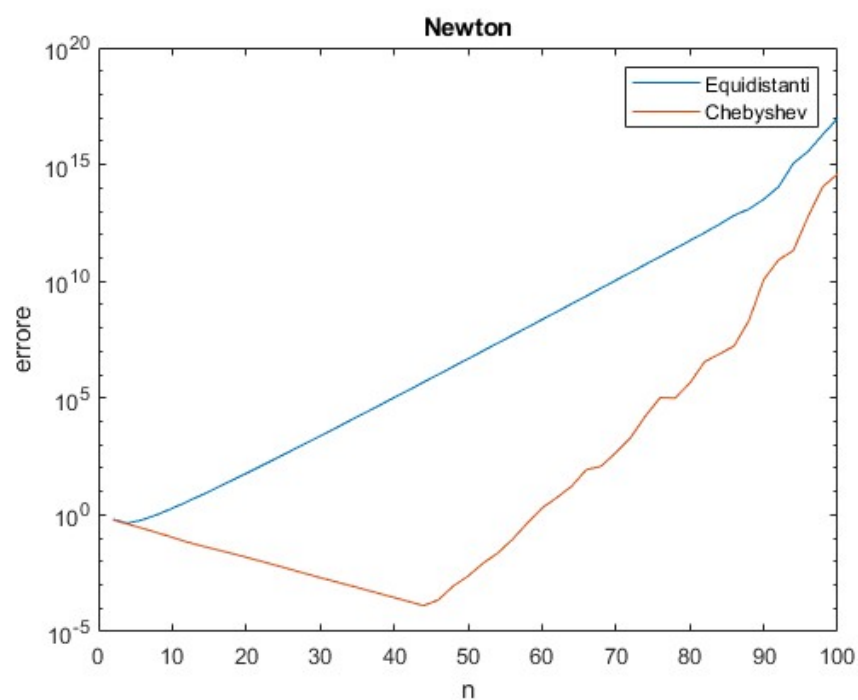
$$f(x) = \frac{1}{1+x^2}, \quad x \in [-5, 5] \quad (9)$$

utilizzando sia le ascisse equidistanti che di Chebyshev, per i polinomi interpolanti di grado `nn=2:2:100`. Graficare l'errore di interpolazione anche per i polinomi interpolanti di Hermite di grado `nn=3:2:99`, sia utilizzando ascisse equidistanti che ascisse di Chebyshev, nell'intervallo considerato.

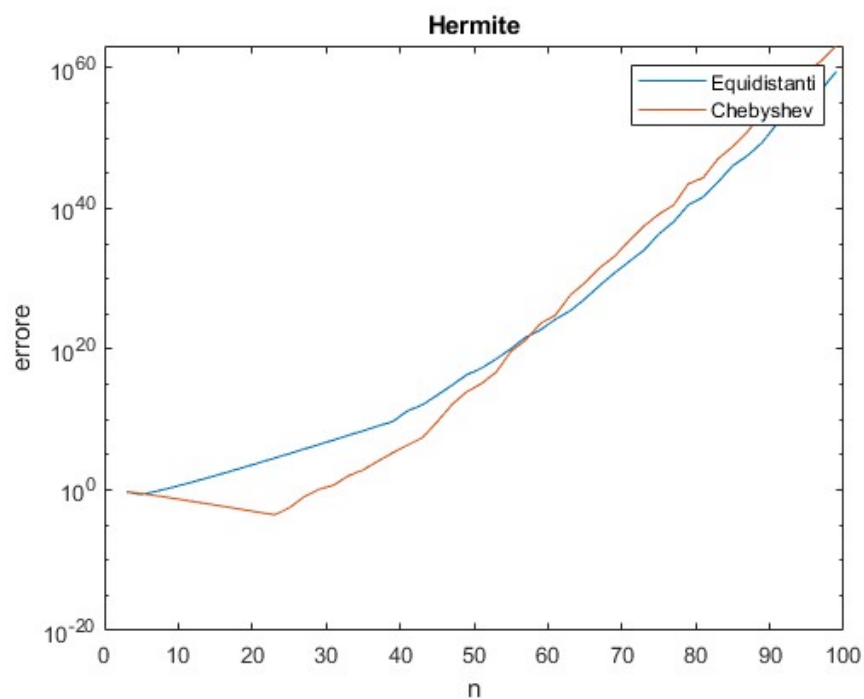
Soluzione: Stimando l'errore con la norma infinito su 10001 punti si ottiene per Lagrange:



Per Newton invece si ottiene:



E similmente per Hermite si ottiene:



Esercizio 22: Costruire una function, `myspline.m`, avente sintassi

`yy = myspline(xi, fi, xx, type)`

dove `type=0` calcola la *spline* cubica interpolante naturale i punti (xi,fi) , e `type~0` calcola quella calcola quella *not-a-knot* (default).

Soluzione: CODICE Matlab spline cubica

```

1 function yy = myspline(xi,fi,xx,type)
2 %
3 % yy=myspline(xi,fi,xx,type)
4 %
5 % Calcolo dei valori della spine interpolante le ascisse
6 %
7 %
8 % Input:
9 % xi: vettori con le ascisse di interpolazione
10 % fi: vettore con i valori della funzione per le ascisse
11 % xx: vettore con i valori su cui calcolare la spline
12 % type: 0 per la spline naturale, diverso da 0 per not-a-knot (default)
13 %
14 % Output:
15 % yy: vettore con i valori della spline calcolati
16 %
17 n=length(xi)-1;
18 if length(unique(xi))~=n+1
19     error("Le ascisse non sono distinte tra loro");
20 end
21 if length(fi)~=n+1
22     error("I vettori xi ed fi devono avere la stessa lunghezza");
23 end
24 h=zeros(1,n);
25 df=fi;
26 if nargin<4
27     type=1;
28 end
29 for j=1:2 %differenze divise
30     for i=n+1:-1:j+1
31         df(i)=(df(i)-df(i-1))/(xi(i)-xi(i-j));
32     end
33 end
34 h=xi(2:n+1)-xi(1:n);
35 phi(1:n-1)=h(1:n-1)./(h(1:n-1)+h(2:n));
36 eps(1:n-1)=h(2:n)./(h(1:n-1)+h(2:n));
37 m(1:n-1)=df(3:n+1)*6;
38 b=phi(2:n-1);
39 a=2*ones(1,n-1);
40 c=eps(1:n-2);
41 if type~=0 %cambio della prima riga e ultima riga per not-a-knot
42     m(1)=m(1)*(1-phi(1));
43     m(n-1)=m(n-1)*(1-eps(n-1));
44     c(1)=c(1)-phi(1);
45     a(1)=a(1)-phi(1);
46     b(n-2)=b(n-2)-eps(n-1);
47     a(n-1)=a(n-1)-eps(n-1);
48 end
49 %risoluzione matrice tridiagonale
50 for i=1:n-2
51     b(i)=b(i)/a(i);

```

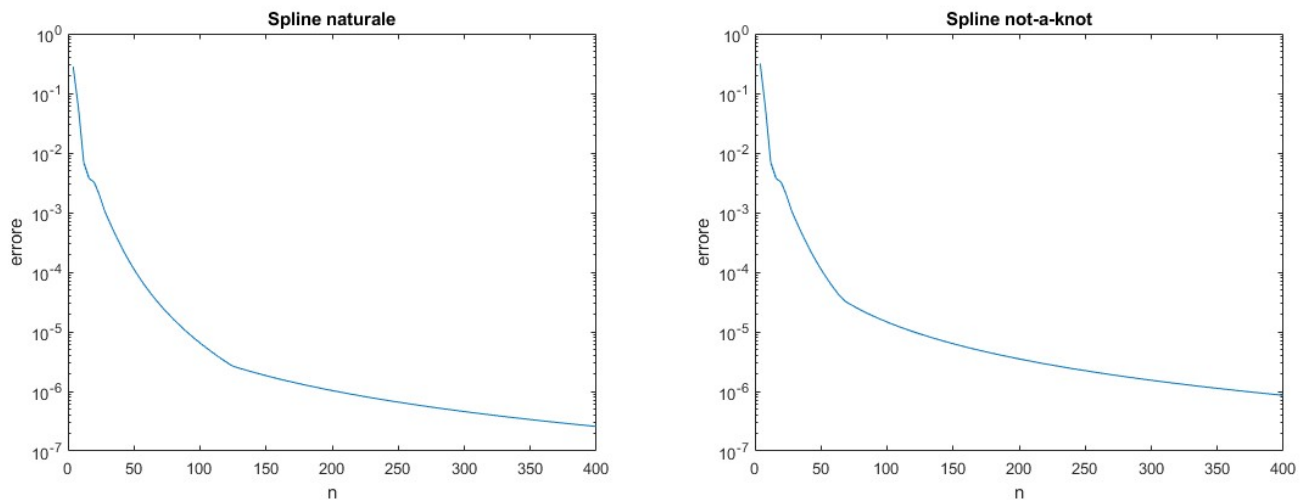
```

52     a(i+1)=a(i+1)-b(i)*c(i);
53     m(i+1)=m(i+1)-b(i)*m(i);
54 end
55 m(n-1)=m(n-1)/a(n-1);
56 for i=n-2:-1:1
57     m(i)=(m(i)-c(i)*m(i+1))/a(i);
58 end
59 %condizioni spline naturale e not-a-knot
60 if type==0
61     m= [0 m 0];
62 else
63     m = [df(3)-m(1)-m(2), m ,df(n+1)-m(n-1)-m(n-2)];
64 end
65 %calcolo di r e q
66 r=zeros(1,n);
67 q=zeros(1,n);
68 for i=1:n
69     r(i)=fi(i)-((h(i)^2)/6)*m(i);
70     q(i)=(fi(i+1)-fi(i))/h(i)-(h(i)/6)*(m(i+1)-m(i)));
71 end
72 %calcolo spline
73 yy=zeros(1,length(xx));
74 for i=1:length(xx)
75     for k=1:n
76         if xx(i)>=xi(k) && xx(i)<=xi(k+1)
77             yy(i)=(((xx(i)-xi(k))^3)*m(k+1)+((xi(k+1)-xx(i))^3)*m(k))/( ...
78                 6*h(k))+q(k)*(xx(i)-xi(k))+r(k);
79             break
80         end
81     end
82 end
83 return
84 end

```

Esercizio 23: Graficare, utilizzando il formato semilogy, l'errore di approssimazione utilizzando le *spline* interpolanti naturale e *not-a-knot* per approssimare la funzione di Runge sull'intervallo $[-5, 5]$, utilizzando una partizione $\Delta = \{-5 = x_0 < \dots < x_n = 5\}$, con ascisse equidistanti e $n=4:4:400$. Utilizzare 10001 punti equispaziati nell'intervallo $[-5, 5]$ per ottenere la stima dell'errore.

Soluzione: Graficando la norma infinito per 10001 per stimare l'errore di interpolazione si ottiene:



E' possibile notare come, rispetto ai polinomi interpolanti, l'errore diminuisce all'aumentare delle ascisse equidistanti.

Esercizio 24: E' noto che un fenomeno fisico evolve come $y = x^n$ con n incognito. Il file `data.mat` contiene 1000 coppie di dati (x_i, y_i) , in cui la seconda componente è affetta da un errore con distribuzione Gaussiana a media nulla e varianza "piccola". Utilizzando un opportuno polinomio di approssimazione ai minimi quadrati, stimare il grado n . Argomentare il procedimento seguito, graficando la norma del residuo rispetto a valori crescenti di n . E richiesto il codice Matlab dell'algoritmo che avrete implementato (potete utilizzare, se lo ritenete opportuno, la function `polyfit` di Matlab).

Soluzione: Dato che l'incognita dell'equazione è l'esponente, conviene innanzitutto riscriverla utilizzando le proprietà dei logaritmi:

$$y = x^n \Rightarrow \ln(y) = \ln(x^n) \Rightarrow \ln(y) = n \ln(x)$$

Così facendo si ottiene una equazione lineare che una volta applicata alle coppie (x_i, y_i) delle misurazioni permette di utilizzare un'approssimazione polinomiale ai minimi quadrati di grado 1 per stimare il valore di n che sarà dato dal coefficiente a_1 . Per meglio trasformare i dati in modo che si avvicinino una retta, conviene usare una rappresentazione in scala semi-logaritmica, si potrebbe infatti approssimare $\ln(x)$ con la retta tangente al punto $x_0 = 1$ ottenendo così: $\ln(y) \approx n(x - 1)$. Un'ulteriore considerazione da fare è quella di scartare le misurazioni y_i negative, essendo le ascisse x_i tutte positive e di conseguenza anche x_i^n , ed inoltre non sarebbe possibile calcolare $\log(y)$.

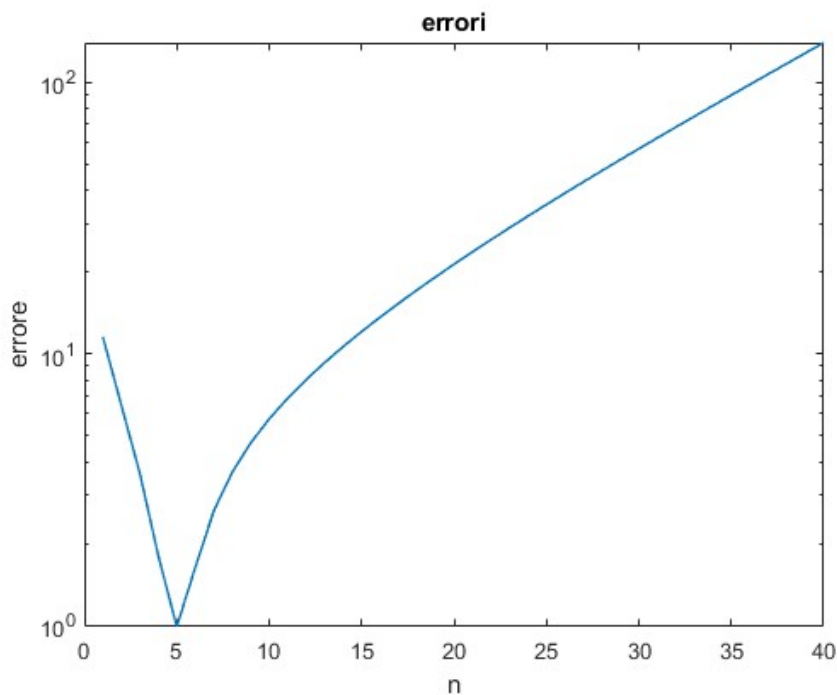
L'algoritmo descritto è stato così implementato:

```

1 function n = estimate(data)
2 %
3 % n = estimate(data)
4 %
5 % Stima il grado di n date m coppie di dati per  $y=x^n$ 
6 %
7 %
8 % Input:
9 % data: matrice mx2 con le misurazioni del problema
10 %
11 % Output:
12 % n: grado stimato del polinomio
13 %
14 fixedData=data;
15 fixedData(fixedData(:,2)<0,:)=[];
16 x=fixedData(:,1);
17 y=log(fixedData(:,2));
18 r=polyfit(x,y,1);
19 n=r(1);
20 return
21 end

```

Si ottiene quindi che il grado n è 5, come si può notare graficando in **semilogy** l'errore (approssimato con la norma 2) del residuo rispetto ad n :



difatti esso è minimo per $n=5$.

Esercizio 25: Costruire una function Matlab che, dato in input n , restituisca i pesi della quadratura della formula di Newton-Cotes di grado n . Tabulare, quindi, i pesi delle formule di grado 1, 2, . . . , 7 e 9 (come numeri razionali).

Soluzione: codice per i pesi di Newton Codes

```

1 function c = pesiNewtCotes(n)
2 %
3 %   c=pesiNewtCotes
4 %
5 %   Calcolo dei pesi della formula di Newton Cotes:
6 %
7 %   Input:
8 %   n: grado della formula
9 %
10 %   Output:
11 %   c: vettore con i pesi
12 %
13 if n<1 || n~=fix(n)
14     error('n deve essere un numero naturale');
15 else
16 c=zeros(1,n);
17 for i=0:n/2
18     den = prod( i - [0:i-1, i+1:n]);
19     coeff = poly([0:i-1 i+1:n]);
20     coeff = [coeff./((n+1):-1:1) 0];
21     num=polyval(coeff,n);
22     c(i+1)=num/den;
23 end
24 for i=n+1:-1:n/2
25     c(i) = c(n+2-i);
26 end
27 return
28 end

```

n	0	1	2	3	4	5	6	7	8	9
1	1/2	1/2								
2	1/3	4/3	1/3							
3	3/8	9/8	9/8	3/8						
4	14/45	64/45	8/15	64/45	14/45					
5	95/288	125/96	125/144	125/144	125/96	95/288				
6	41/140	54/35	27/140	68/35	27/140	54/35	41/140			
7	1073/3527	810/559	343/640	649/536	649/536	343/640	810/559	1073/3527		
9	130/453	1374/869	243/2240	5287/2721	704/1213	704/121	5287/2721	243/2240	1374/869	130/453

Esercizio 26: Scrivere una function Matlab,

[If,err] = composita(fun, a, b, k, n)

che implementi la formula composta di Newton-Cotes di grado k su $n+1$ ascisse equidistanti, con n multiplo pari di k , in cui:

- fun è la funzione integranda (che accetta input vettoriali);
- $[a,b]$ è l'intervallo di integrazione;
- k, n come su descritti;
- If è l'approssimazione dell'integrale ottenuta;
- err è la stima dell'errore di quadratura.

Soluzione: Calcolo della formula composta di Newton-Cotes di grado k su $n+1$ ascisse equidistanti.

```

1 function [If,err] = composita(fun, a, b, k, n)
2 x=linspace(a,b,n+1);
3 y=fun(x);
4 h=(b-a)/n;
5 c=pesiNewtCotes(k);
6 If=0;
7 for i=1:k:n+1-k
8     If=If+sum(y(i:i+k).*c);
9 end
10 If=h*If;
11 IfH=0;
12 for i=1:2*k:n+1-2*k
13     IfH=IfH+sum(y(i:2:i+2*k).*c);
14 end
15 IfH=2*h*IfH;
16 if mod(k,2)==1
17     u=1;
18 else
19     u=2;
20 end
21 err=abs(IfH-If)/(2^(k+u)-1);

```

Esercizio 27: Utilizzare la function `composita` per ottenere l'approssimazione dell'integrale

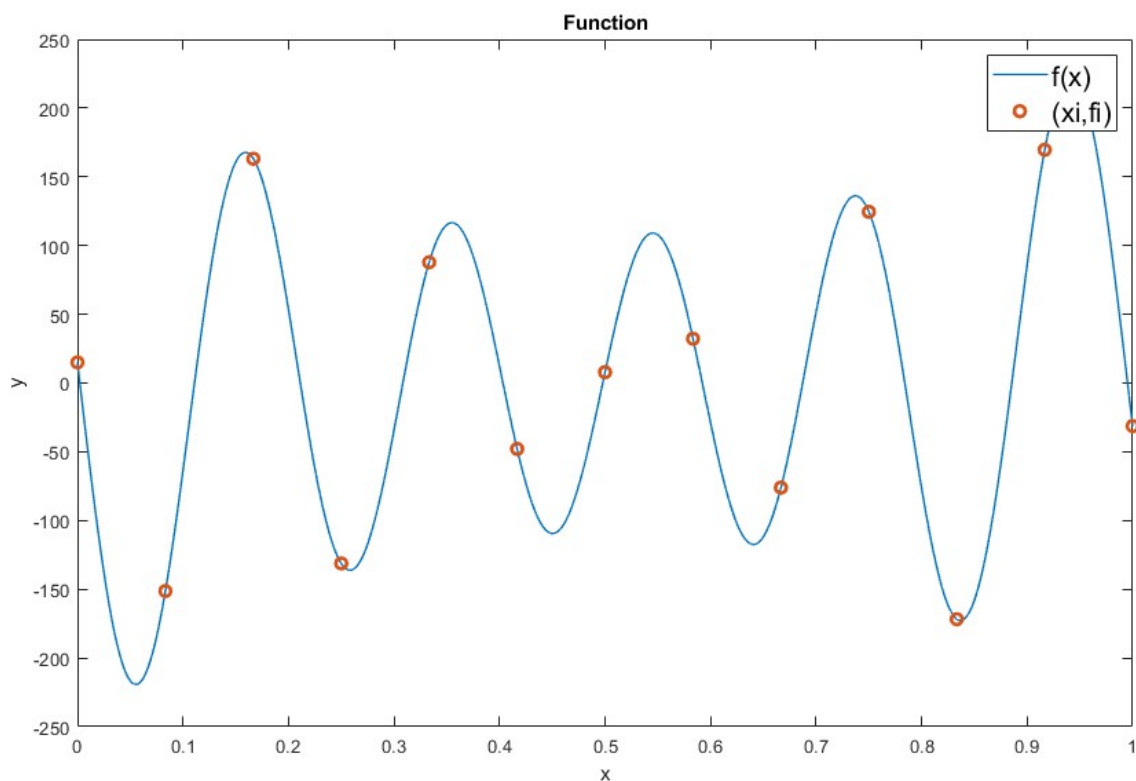
$$\int_0^1 \left(\sum_{i=1}^5 i \cos(2\pi i x) - e^i \sin(2(\pi i + 0.1)x) \right) dx$$

con le formule composite di Newton-Cotes di grado $k=1, 2, 3, 6$. Per tutte, utilizzare $n=12$.

Soluzione: Utilizzando le formule composite di Newton-Cotes con $n=12$ per approssimare l'integrale si vanno ad ottenere i seguenti risultati:

k	valore	errore
1	-0.092598047616972	0.192379444267582
2	-0.284977491884554	0.034967622162555
3	0.114030927423084	0.029823785706593
6	-0.702525851480416	0.002428474323110

Si può notare come i risultati risultano essere imprecisi e discordanti tra loro, infatti andando a graficare la funzione in esame negli estremi dell'intervallo di integrazione



possiamo notare che oscilla frequentemente, rendendo le formule più inaccurate per le ascisse considerate. Aumentando il numero di quest'ultime si può ridurre ulteriormente l'errore, d'altronde si ha:

$$E_k^{(n)} = v_k f^{(k+u)}(\xi) \left(\frac{b-a}{k} \right) \left(\frac{b-a}{n} \right)^{(k+u)} \text{ con } \xi \in [a, b]$$

che sappiamo tende a 0 per n che tende ad infinito, altrimenti si possono usare pure le formule adattive.

Esercizio 28: Implementare la formula composta adattativa di Simpson.**Soluzione:** Calcolo della formula adattativa di Simpson

```

1 function [I2,vf] = adapsim(a,b,f,tol,fa,f1,fb)
2 %
3 % [I2,vf] = adapsim(a,b,f,tol)
4 %
5 % Calcola la formula adattativa di Simpson
6 %
7 % Input:
8 % a,b: estremi intervallo di integrazione
9 % f: function funzione integranda
10 % tol: tolleranza richiesta
11 % Output:
12 % I2: approssimazione ottenuta
13 % vf: valutazioni funzionali
14 %
15 if a==b
16     I2=0;
17     return
18 elseif a>b
19     error('intervallo non corretto');
20 elseif tol<0
21     error('tolleranza negativa');
22 end
23 x1=(a+b)/2;
24 vf=0;
25 if nargin==4
26     fa=feval(f,a);
27     fb=feval(f,b);
28     f1=feval(f,x1);
29     vf=3;
30 end
31 h=(b-a)/6;
32 I1=h*(fa+4*f1+fb);
33 x2=(a+x1)/2;
34 x3=(x1+b)/2;
35 f2=feval(f,x2);
36 f3=feval(f,x3);
37 I2=.5*h*(fa+4*f2+2*f1+4*f3+fb);
38 vf=vf+2;
39 e=abs(I2-I1)/15;
40 if e>tol
41     [left,vf1]=adapsim(a,x1,f,tol/2,fa,f2,f1);
42     [right,vf2]=adapsim(x1,b,f,tol/2,f1,f3,fb);
43     I2=left+right;
44     vf=vf+vf1+vf2;
45 end
46 return
47 end

```

Esercizio 29: Implementare la formula composta adattativa di Newton-Cotes di grado $k=4$.

Soluzione: Calcolo della formula adattativa di Newton-Cotes di grado $k=4$.

```

1 function [I2,vf] = adapquad(a,b,f,tol,fa,f1,f2,f3,fb)
2 %
3 %   [I2,vf] = adapquad(a,b,f,tol)
4 %
5 %   Calcola la formula adattativa di Simpson
6 %
7 %   Input:
8 %   a,b: estremi intervallo di integrazione
9 %   f: function funzione integranda
10 %   tol: tolleranza richiesta
11 %   Output:
12 %   I2: approssimazione ottenuta
13 %   vf: valutazioni funzionali
14 %
15 if a==b
16     I2=0;
17     return
18 elseif a>b
19     error('intervallo non corretto');
20 elseif tol<0
21     error('tolleranza negativa');
22 end
23 vf=0;
24 x2=(a+b)/2;
25 x1=(a+x2)/2;
26 x3=(x2+b)/2;
27 if nargin==4
28     fa=feval(f,a);
29     fb=feval(f,b);
30     f1=feval(f,x1);
31     f2=feval(f,x2);
32     f3=feval(f,x3);
33     vf=5;
34 end
35 h=(b-a)/180;
36 I1=h*(14*fa+64*f1+24*f2+64*f3+14*fb);
37 x4=(a+x1)/2;
38 x5=(x1+x2)/2;
39 x6=(x2+x3)/2;
40 x7=(x3+b)/2;
41 f4=feval(f,x4);
42 f5=feval(f,x5);
43 f6=feval(f,x6);
44 f7=feval(f,x7);
45 I2=.5*h*(14*fa+64*f4+24*f1+64*f5+28*f2+64*f6+24*f3+64*f7+14*fb);
46 vf=vf+4;
47 e=abs(I2-I1)/63;
48 if e>tol
49     [left,vf1]=adapquad(a,x2,f,tol,fa,f4,f1,f5,f2);
50     [right,vf2]=adapquad(x2,b,f,tol,f2,f6,f3,f7,fb);
51     I2=left+right;
52     vf=vf+vf1+vf2;
53 end
54 return
55 end

```

Esercizio 30: Confrontare le formule adattative degli ultimi due esercizi, tabulando il numero di valutazioni funzionali effettuate, rispetto alla tolleranza $\text{tol} = 1\text{e-}2, 1\text{e-}3, \dots, 1\text{e-}9$, per ottenere l'approssimazione dell'integrale

$$\int_{10^{-5}}^1 x^{-1} \cos(\log(x^{-1})) dx \equiv \sin(\log(10^5))$$

Costruire un'altra tabella, in cui si tabula l'errore vero (essendo l'integrale noto, in questo caso) rispetto a tol .

Soluzione: Tabulando le valutazioni funzionali si ottiene:

tol	valutazioni funzionali n=2	valutazioni funzionali n=4
1e-2	201	113
1e-3	333	121
1e-4	605	129
1e-5	1061	137
1e-6	1869	145
1e-7	3277	265
1e-8	5921	345
1e-9	10589	473

Mentre tabulando l'errore effettivo si ottiene:

tol	n=2		n=4	
	valore	errore	valore	errori
1e-2	-0.869425809078715	2.935280256943784e-04	-0.818445828479865	0.050686452573156
1e-3	-0.869589504976433	4.572239234117426e-04	-0.863917726701441	0.005214554351580
1e-4	-0.869158635812636	2.635475961443312e-05	-0.868851170727444	2.811103255769831e-04
1e-5	-0.869134625866697	2.344813675669855e-06	-0.869126106965213	6.174087807786499e-06
1e-6	-0.869132581815646	3.007626251383400e-07	-0.869134425591871	2.144538849502276e-06
1e-7	-0.869132317617779	3.656475811020243e-08	-0.869132234992719	4.606030246101511e-08
1e-8	-0.869132284264119	3.211098165145643e-09	-0.869132276557698	4.495323446818134e-09
1e-9	-0.869132281362861	3.098394874001542e-10	-0.869132278121886	2.931135223427361e-09

Da cui si può notare che le valutazioni funzionali per Simpson sono molto maggiori per le formule adattative con $n=4$, tuttavia con $n=2$ si riesce a raggiungere un errore inferiore rispetto all'altra formula a parità di tolleranza.