



Elaborato di
Calcolo Numerico
Anno Accademico 2021/2022

John Smith - 1234567 - *john.smith@stud.unifi.it*
Nice Guy - 9876543 - *nice.guy@stud.unifi.it*

July 5, 2022

Contents

1	Capitolo 1	4
1.1	Esercizio 1	4
1.2	Esercizio 2	4
1.3	Esercizio 3	5
2	Capitolo 2	6
2.1	Esercizio 4	6
2.2	Esercizio 5	7
2.3	Esercizio 6	10
2.4	Esercizio 7	11
3	Capitolo 3	13
3.1	Esercizio 8	13
3.2	Esercizio 9	15
3.3	Esercizio 10	18
3.4	Esercizio 11	18
3.5	Esercizio 12	18
3.6	Esercizio 13	20
3.7	Esercizio 14	20
4	Capitolo 4	21
4.1	Esercizio 15	21
4.2	Esercizio 16	21
4.3	Esercizio 17	21
4.4	Esercizio 18	21
4.5	Esercizio 19	21
4.6	Esercizio 20	21
5	Capitolo 5	22
5.1	Esercizio 21	22
5.2	Esercizio 22	22
5.3	Esercizio 23	22
5.4	Esercizio 24	22
5.5	Esercizio 25	22
5.6	Esercizio 26	23
5.7	Esercizio 30	23
5.8	Esercizio 28	23
5.9	Esercizio 29	23
5.10	Esercizio 30	23
6	Codici ausiliari	24
6.1	Esercizio 6	24
6.2	Esercizio 7	24
6.3	Esercizi 8 e 9	25
6.4	Esercizio 15	25
6.5	Esercizio 16	25
6.6	Esercizio 18	26
6.7	Esercizio 19	26
6.8	Esercizio 20	27
6.9	Esercizio 21	27
6.10	Esercizio 22	27
6.11	Esercizio 23	28
6.12	Esercizio 24	28
6.13	Esercizio 25	28

List of Figures

1	iterazioni richieste	11
2	iterazioni richieste	12

List of Tables

1	valori approssimati con i metodi di Newton, secanti e Steffensen	10
2	valori approssimati con i metodi di Newton, secanti e Steffensen	11

1 Capitolo 1

1.1 Esercizio 1

Verificare che,

$$\frac{-f(x+2h) + 8f(x+h) - 8f(x-h) + f(x-2h) + O(h^4)}{12h} = f'(x) + O(h^4)$$

Soluzione:

Per verificare la equivalenza ci servirà applicare sviluppo di Taylor per tutti i funzioni. La funzione di Taylor definita così:

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0)^2 + \dots + \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n + O((x - x_0)^n)$$

Calcoliamo le sviluppi di Taylor per tutti funzioni che sono presenti in formula da verificare. A variabile x_0 noi assumiamo il valore x , quindi $x_0 = x$. Allora:

$$f(x+2h) = f(x) + f'(x)(x+2h-x) + O(x+2h-x) = f(x) + 2hf'(x) + O(h)$$

$$f(x+h) = f(x) + f'(x)(x+h-x) + O(x+h-x) = f(x) + hf'(x) + O(h)$$

$$f(x-h) = f(x) + f'(x)(x-h-x) + O(x-h-x) = f(x) - hf'(x) + O(h)$$

$$f(x-2h) = f(x) + f'(x)(x-2h-x) + O(x-2h-x) = f(x) - 2hf'(x) + O(h)$$

Se mettere tutto insieme otteniamo:

$$\begin{aligned} & \frac{-(f(x) + 2hf'(x)) + 8(f(x) + hf'(x)) - 8(f(x) - hf'(x)) + (f(x) - 2hf'(x)) + O(h^4)}{12h} = \\ & = \frac{-\cancel{f(x)} - 2hf'(x) + \cancel{8f(x)} + 8hf'(x) - \cancel{8f(x)} + 8hf'(x) + \cancel{f(x)} - 2hf'(x) + O(h^4)}{12h} = \\ & = \frac{12hf'(x) + O(h^4)}{12h} = f'(x) + O(h^4) \end{aligned}$$

1.2 Esercizio 2

Calcolare, motivandone i passaggi, la precisione di macchina della doppia precisione dello standard IEEE. Confrontare questa quantità con quanto ritornato dalla variabile `eps` di Matlab, commentando a riguardo.

Soluzione:

Dato un generico numero reale $x \in I$, con I sottoinsieme della retta reale, e dato \mathbf{M} come insieme dei numeri di macchina che appartengono a sottoinsieme della retta reale I , sappiamo dalla teoria che il numero di elementi di \mathbf{M} è finito. Pertanto è necessario definire una funzione $fl : IV\mathbf{M}$, se $x \in I$ e $x \neq 0$ con $fl(x) = x(1 + \varepsilon_x)$, dove ε_x è l'errore relativo della funzione tale che $\varepsilon_x \leq u$,

$$u = \begin{cases} 2^{1-m}, & \text{per troncamento} \\ \frac{1}{2} * 2^{1-m}, & \text{per arrotondamento} \end{cases}$$

Poiché lo studio riguarda la doppia precisione, la mantissa m è uguale a 53. Per confrontare questi valori con l'`eps` di Matlab, è necessario eseguire il seguente codice. Le variabili `truncated` e `rounded` avranno rispettivamente i risultati del troncamento e dell'arrotondamento:

```
1 >> a=eps
2 a = 2.2204e-16
3 >> truncated = 2^(1-53)
4 truncated = 2.2204e-16
5 >> rounded = (1/2)*(2^(1-53))
6 rounded = 1.1102e-16
```

Dai precedenti valori, è chiaro che l'`epsilon` di macchina combacia con il valore della precisione usato per il troncamento, mentre è il doppio esatto di quello per l'arrotondamento.

1.3 Esercizio 3

Eseguire il seguente script Matlab:

```
1 format long e
2 (1+(1e-14-1))*1e14
```

Spiegare i risultati ottenuti.

Soluzione:

```
1 >> format long e
2 >> (1+(1e-14-1))*1e14
3 ans = 9.992007221626409e-01
```

Il risultato è dovuto alla cosiddetta **cancellazione numerica**: i due addendi sono rappresentati in macchina con piena accuratezza, a causa però del malcondizionamento del problema ($k \gg 1$), il risultato non è corretto.

2 Capitolo 2

2.1 Esercizio 4

Scrivere una function Matlab, *radice(x)* che, avendo in ingresso un numero x non negativo, ne calcoli la radice quadrata utilizzando solo operazioni algebriche elementari, con la massima precisione possibile. Confrontare con la function *sqrt* di Matlab per 20 valori di s , equispaziati logaritmicamente nell'intervallo $[1e-10, 1e10]$, evidenziando che si è ottenuta la massima precisione possibile.

Soluzione:

```
1 function answer = radice(x)
2     % answer = radice(x)
3     % This method is called Babilons method
4     % It gets some value in input and apply the Babilons method
5     % until I needed precision, but we need maximum precision possible,
6     % so I apply this method until I get two equal values
7     % one before another and this will mean this method arrived at a point,
8     % in which value not changes, so it is a maximum precision.
9     % INPUT:  x      — Is a value of which we want to get square root
10    % OUTPUT: answer — This is the square root of the input
11    format long e;
12
13    if x < 0
14        error('Wrong value in input! Should be not less then zero!');
15    end
16
17    answer = x / 2;
18    guess = x;
19
20    while not(answer == guess)
21        guess = answer;
22        answer = (guess + (x / guess)) / 2;
23    end
24
25 end
26
27 for i = 1:0.5:10
28     s = 1e-10 + 1e10 * log10(i);
29     custom = radice(s);
30     native = sqrt(s);
31
32     if abs(custom - native) >= eps(abs(native))
33
34         if abs(custom - native) > eps(abs(native))
35             diff = abs(custom - native);
36             fprintf([ ...
37                 'result is not equal for %.' int2str(abs(int16(log10(eps(s)))) + 1)
38                 ...
39                 'd, with values custom=.' int2str(abs(int16(log10(eps(custom)))) +
40                 20) ...
41                 'd and native=.' int2str(abs(int16(log10(eps(native)))) + 20) ...
42                 'd, with diff=.' int2str(abs(int16(log10(eps(diff)))) + 20) 'd\n'
43                 ...
44                 ], s, custom, native, diff);
45         elseif abs(custom - native) == eps(abs(native))
46             fprintf([ ...
47                 'Difference between this numbers are equal to exponent ' ...
48                 'for this numbers, and equals to %.30d\n'...
```

```

46         ], eps(abs(native)));
47     end
48
49 end
50
51 fprintf([ ...
52     'custom=%. ' int2str(abs(int16(log10(eps(custom)))) + 1) ...
53     'd\nnative=%. ' int2str(abs(int16(log10(eps(native)))) + 1) ...
54     'd\n\n'
55     ], custom, native);
56
57 end

```

2.2 Esercizio 5

Scrivere function Matlab distinte che implementino efficientemente i seguenti metodi per la ricerca degli zeri di una funzione $f(x)$:

- il metodo di Newton;
- il metodo delle secanti;
- il metodo di Steffensen:

$$x_{n+1} = x_n - \frac{f(x_n)^2}{f(x_n + f(x_n)) - f(x_n)}, \quad n=0,1,\dots$$

Per tutti i metodi, utilizzare come criterio di arresto

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

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

Soluzione:

- Metodo di Newton

```

1 function [x, i] = newton(f, f1, x0, tolx, maxit)
2     % [x, i] = newton(f, f1, x0, tolx, maxit)
3     % Newton's method, used to calculate a root of the equation f(x)=0
4     % Input:  f      — function that we can derive
5             f1     — derived function of the 'f'
6             x0     — initial approximation
7             tolx   — tolerance
8             maxit  — maximum number of iterations (default = 100)
9     % Output: x      — the approximation of the root function
10            i       — number of iterations
11    % CHECK THIS: bisezione, corde, secanti, aitken, newtonmod
12
13    format long e;
14
15    if nargin < 4
16        error('Number of arguments must be at least 4!');
17    elseif nargin == 4
18        maxit = 100;
19    end
20
21    if tolx < eps
22        error('Tolerance cannot be checked!');

```

```

23     end
24
25     precisionEnough = false;
26
27     for i = 1:maxit
28         fx = feval(f, x0);
29         flx = feval(f1, x0);
30
31         if flx == 0
32             warning(['Value of f1 on iteration ' int2str(i) ' is zero!']);
33             break;
34         end
35
36         x = x0 - (fx / flx);
37
38         precisionEnough = abs(x - x0) <= tolx * (1 + abs(x0));
39
40         if precisionEnough
41             break;
42         end
43
44         x0 = x;
45
46     end
47
48     if ~precisionEnough
49         warning(['Failed to converge in ' maxit ' iterations!']);
50     end
51
52 end

```

- Metodo delle secanti

```

1 function [x, i] = secanti(f, x0, x1, tolx, maxit)
2     % [x, i] = secanti(f, x0, x1, tolx, maxit)
3     % Method of the secant, used to calculate a root of the equation f(x)=0
4     % Input:  f      — function that we can derive
5             x0      — initial approximation
6             x1      — second initial approximation
7             tolx    — tolerance
8             maxit   — maximum number of iterations (default = 100)
9     % Output: x      — the approximation of the root function
10            i       — number of iterations
11     % CHECK THIS: bisezione, corde, secanti, aitken, newtonmod
12
13     format long e;
14
15     if nargin < 4
16         error('Number of arguments must be at least 4!');
17     elseif nargin == 4
18         maxit = 100;
19     end
20
21     precisionEnough = false;
22     i = 0;
23     f0 = feval(f, x0);
24
25     for i = 1:maxit
26         f1 = feval(f, x1);

```



```

27     df1 = (f1 - f0) / (x1 - x0);
28
29     if df1 == 0
30         warning(['Delta of f on iteration ' int2str(i) ' is zero!']);
31         break;
32     end
33
34     x0 = x1;
35     x1 = x1 - (f1 / df1);
36     f0 = f1;
37     precisionEnough = abs(x1 - x0) <= tolx * (1 + abs(x0));
38
39     if precisionEnough
40         break;
41     end
42
43 end
44
45 if ~precisionEnough
46     warning(['Failed to converge in ' maxit ' iterations!']);
47 end
48
49 x = x1;
50 end

```

- Metodo di Steffensen

```

1 function [x, i] = steffensen(f, x0, tolx, maxit)
2     % [x, i] = steffensen(f, x0, tolx, maxit)
3     % Steffensen's method, used to calculate a root of the equation f(x)=0
4     % Input:  f      — a fixed point iteration function
5             x0      — initial guess to the fixed point
6             tolx    — tolerance
7             maxit    — maximum number of iterations (default = 100)
8     % Output: x      — the approximation of the root function
9             i        — number of iterations
10    % CHECK THIS: bisezione, corde, secanti, aitken, newtonmod
11
12    format long e;
13
14    if nargin < 3
15        error('Number of arguments must be at least 3!');
16    elseif nargin == 3
17        maxit = 100;
18    end
19
20    precisionEnough = false;
21
22    for i = 1:maxit
23        % get ready to do a large, but finite, number of iterations.
24        % This is so that if the method fails to converge, we won't
25        % be stuck in an infinite loop.
26        fx = feval(f, x0); % calculate the next two guesses for the fixed point.
27        f_fx_x = feval(f, fx + x0);
28
29        if (f_fx_x - fx) == 0
30            warning(['Distance between two guesses on iteration ' int2str(i) ' is
31                    zero!']);
32            break;

```

```

32     end
33
34     x = x0 - (fx^2 / (f_fx_x - fx));
35     % use Aitken's delta squared method to
36     % find a better approximation to x0.
37     precisionEnough = abs(x - x0) <= tolx * (1 + abs(x0));
38
39     if precisionEnough
40         break; % if we are, stop the iterations, we have our answer.
41     end
42
43     x0 = x;
44
45 end
46
47 if ~precisionEnough
48     warning(['Failed to converge in ' int2str(maxit) ' iterations!']);
49 end
50
51 end

```

2.3 Esercizio 6

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

$$f(x) = x - \cos\left(\frac{\pi}{2}x\right),$$

per $tol = 10^{-3}, 10^{-6}, 10^{-9}, 10^{-12}$, partendo da $x_0 = 1$ (e $x_1 = 0.99$ per il metodo delle secanti). Tabulare i risultati, in modo da confrontare le iterazioni richieste da ciascun metodo. Commentare il relativo costo computazionale, in termini di valutazioni funzionali richieste.

Soluzione:

Eseguendo lo script es6.msi ottengono i risultati contenuti nella tabella 1 e nella figura 1. Come si può notare, il metodo di newton e il metodo delle secanti convergono molto più rapidamente del metodo di bisezione e del metodo delle corde.

Metodo	newton	secanti	steffensen
tolleranza= 10^{-3}	5.946116463605413e-01	5.946184776717105e-01	5.946116811419248e-01
tolleranza= 10^{-6}	5.946116440568356e-01	5.946116440568420e-01	5.946116440568371e-01
tolleranza= 10^{-9}	5.946116440568356e-01	5.946116440568356e-01	5.946116440568356e-01
tolleranza= 10^{-12}	5.946116440568356e-01	5.946116440568356e-01	5.946116440568356e-01

Table 1: valori approssimati con i metodi di Newton, secanti e Steffensen

Costo computazionale

È possibile valutare i costi computazionali dei algoritmi verificando il numero di funzioni richiamate all'interno dei codici, mediante feval:

- Newton: $2i$, $i = 1 : maxit$;
- Secanti: $1 + i$, $i = 1 : maxit$;
- Steffensen: $2i$, $i = 1 : maxit$;

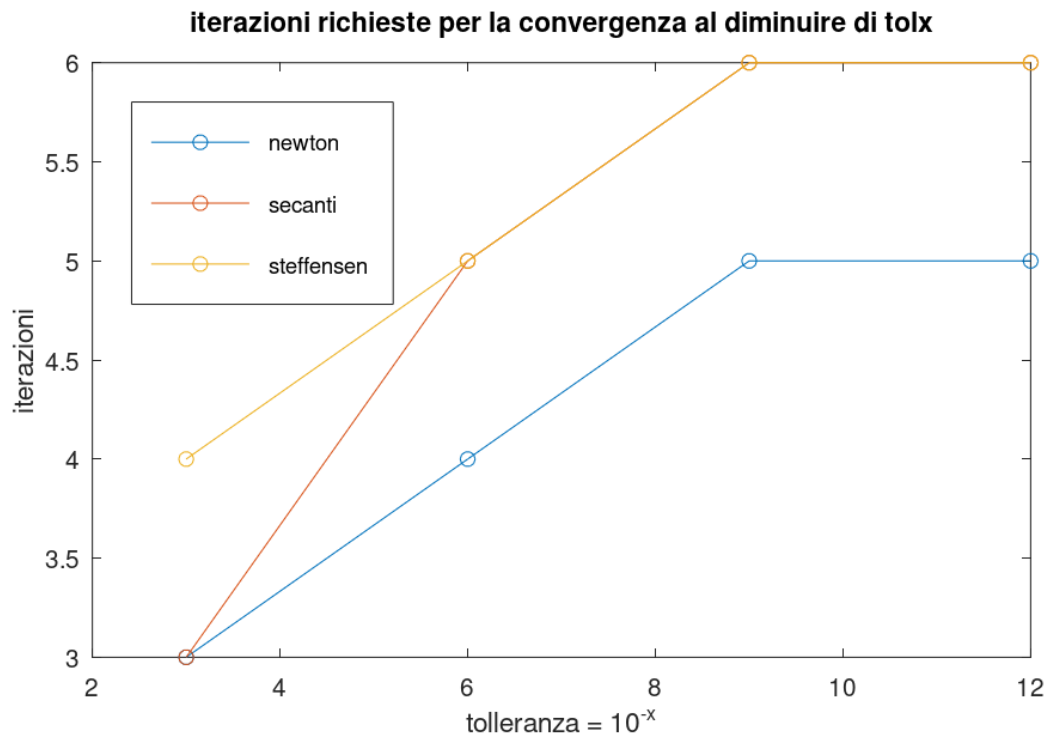


Figure 1: iterazioni richieste

2.4 Esercizio 7

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

$$f(x) = \left[x - \cos\left(\frac{\pi}{2}x\right) \right]^3,$$

per $tol = 10^{-3}, 10^{-6}, 10^{-9}, 10^{-12}$, partendo da $x_0 = 1$ (e $x_1 = 0.99$ per il metodo delle secanti). Tabulare i risultati, in modo da confrontare le iterazioni richieste da ciascun metodo. Commentare i risultati ottenuti.

Soluzione:

Eseguendo lo script es7.m si ottengono i risultati contenuti nella tabella 2 e nella figura 2. Come si può notare, il metodo di newton e il metodo delle secanti convergono molto più rapidamente del metodo di bisezione e del metodo delle corde.

Metodo	newton	secanti	steffensen
tolleranza= 10 ⁻³	5.969479343078770e-01	5.991437227787725e-01	5.973965526716639e-01
tolleranza= 10 ⁻⁶	5.946140179818806e-01	5.946156634766230e-01	5.946143706333854e-01
tolleranza= 10 ⁻⁹	5.946116464662755e-01	5.946116487688006e-01	5.946130329177074e-01
tolleranza= 10 ⁻¹²	5.946116440592810e-01	5.946116440610053e-01	5.946130329177074e-01

Table 2: valori approssimati con i metodi di Newton, secanti e Steffensen

In iterazione 38 la funzione con metodo di Steffensen fallisce con valori di tolleranza 10^{-9} e 10^{-12} . Ed infatti, si vede che in questi valori la figura 2 rimane costante rispetto iterazioni. Questo fallimento dato da divisione a zero.

Costo computazionale

È possibile valutare i costi computazionali dei algoritmi verificando il numero di funzioni richiamate all'interno dei codici, mediante feval:

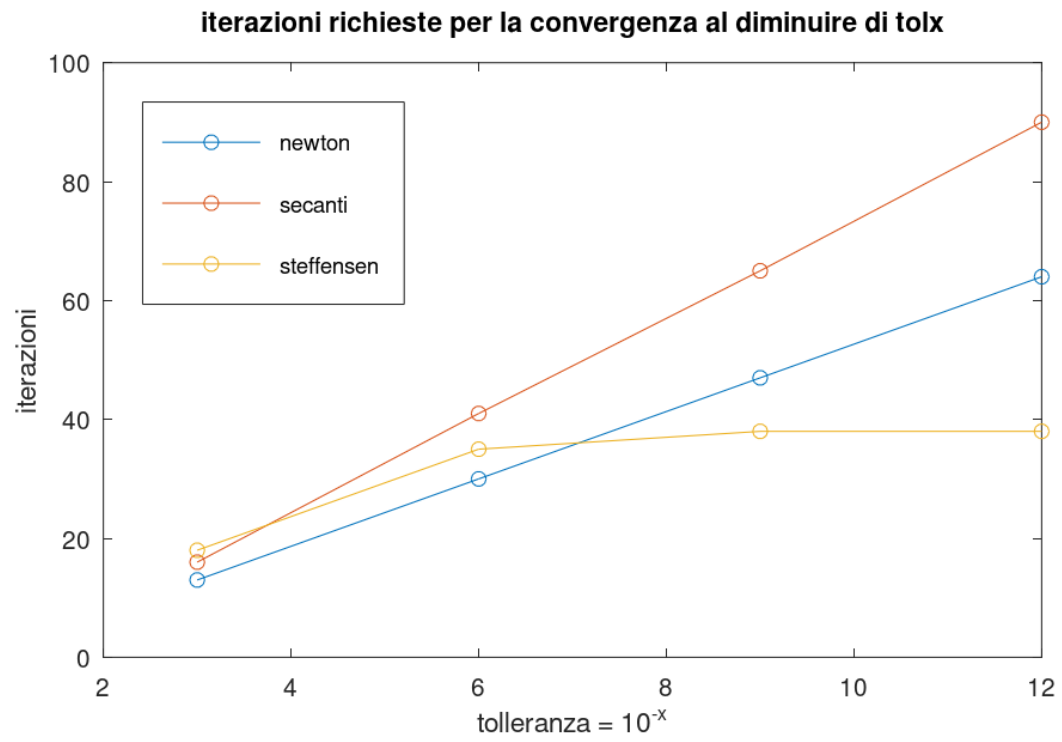


Figure 2: iterazioni richieste

- Newton: $2i, i = 1 : maxit$;
- Secanti: $1 + i, i = 1 : maxit$;
- Steffensen: $2i, i = 1 : maxit$;

3 Capitolo 3

3.1 Esercizio 8

Scrivere una function Matlab,

```
1 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:

```
1 function x = mialu(A, b)
2     % Syntax: x = mialu(A, b)
3     % Method of the secant, used to calculate a root of the equation f(x)=0
4     % Input:  A    — a matrix representation of the equations with unknown variables
5     %         b    — result of unknown variables
6     % Output: x    — array of unknown variables
7
8     % Calculate L and U matrixes
9     n = length(A);
10    L = zeros(n);
11    U = zeros(n);
12    % P = eye(n);
13
14    for k = 1:n
15        % find the entry in the left column with the largest abs value (pivot)
16        [~, r] = max(abs(A(k:end, k)));
17        r = n - (n - k + 1) + r;
18
19        A([k r], :) = A([r k], :);
20        % P([k r], :) = P([r k], :);
21        L([k r], :) = L([r k], :);
22        b([k r], :) = b([r k], :);
23
24        % from the pivot down divide by the pivot
25        L(k:n, k) = A(k:n, k) / A(k, k);
26
27        U(k, 1:n) = A(k, 1:n);
28        A(k + 1:n, 1:n) = A(k + 1:n, 1:n) - L(k + 1:n, k) * A(k, 1:n);
29
30    end
31
32    U(:, end) = A(:, end);
33
34    x = zeros(n, 1);
35    y = zeros(n, 1);
36
37    % calculate answers for Ly = b
38    for i = 1:1:n
39        alpha = 0;
40
41        for k = 1:1:i
42            alpha = alpha + L(i, k) * y(k);
43        end
44
45        y(i) = b(i) - alpha;
46    end
```

```

47
48 % calculate answers for Ux = y
49 for i = n:-1:1
50     alpha = 0;
51
52     for k = i + 1:n
53         alpha = alpha + U(i, k) * x(k);
54     end
55
56     x(i) = (y(i) - alpha) / U(i, i);
57 end
58
59 end

```

Per verificare che soluzioni sono giusti è stata creata la funzione testMatrixSolution.m che moltiplica valori di soluzione con la riga e somma tutti valori ottenuti dopodiché stampa questo valore, che deve essere uguale allo valore di stessa riga di b .

```

1  >> matrixDimension = 4;
2  >> A = round(10 * rand(matrixDimension))
3  A =
4
5      1     4     6     4
6      3    10     2     8
7      8     7     7     0
8     10     4     6     1
9
10 >> b = round(10 * rand(matrixDimension, 1))
11 b =
12
13     8
14     3
15    10
16     6
17
18 >> solutions = es8_lu(A, b)
19 solutions =
20
21    -0.3681
22     0.4884
23     1.3609
24    -0.4377
25
26 >> testMatrixSolution(A,solutions)
27 8
28 3.0000
29 10.0000
30 6.0000
31 >> ;
32 >> ;
33 >> ;
34 >> matrixDimension = 3;
35 >> A = round(10 * rand(matrixDimension))
36 A =
37
38     9     5     5
39     9     0     3
40     2     1     2
41

```

```

42 >> b = round(10 * rand(matrixDimension, 1))
43 b =
44
45     1
46     8
47     7
48
49 >> solutions = es8_lu(A, b)
50 solutions =
51
52    -1.4762
53    -4.2381
54     7.0952
55
56 >> testMatrixSolution(A,solutions)
57 1
58 8
59 7
60 >>

```

3.2 Esercizio 9

Scrivere una function Matlab,

```
1 function x = mialdl(A, b)
```

che, dati in ingresso una matrice sdp 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:

```

1 function x = mialdl(A, b)
2     % Syntax: x = mialdl(A, b)
3     % Method of the secant, used to calculate a root of the equation f(x)=0
4     % Input:  A    — a matrix representation of the equations with unknown variables
5     %          It is assumed that A is symmetric and postive definite.
6     %          b    — result of unknown variables
7     % Output: x    — array of unknown variables
8
9     try
10        % Checking if matrix is SPD with if is not most efficient way.
11        % But still can be used by next syntax:
12        % `~issymmetric(A) || ~all(eig(A) > 0)`
13        % According to
14        % [this](https://www.mathworks.com/help/matlab/math/determine-whether-matrix-is-
15        %   positive-definite.html)
16        % article most efficient way is to use Cholesky Factorization.
17        chol(A);
18    catch
19        error('The matrix provided is not Symmetric Positive Definite!')
20    end
21
22    % Figure out the size of A.
23    n = size(A, 1);
24    % The main loop. See Golub and Van Loan for details.
25    L = zeros(n, n);
26    for j = 1:n,

```

```

27
28     if (j > 1),
29         v(1:j - 1) = L(j, 1:j - 1) .* d(1:j - 1);
30         v(j) = A(j, j) - L(j, 1:j - 1) * v(1:j - 1)';
31         d(j) = v(j);
32
33         if (j < n),
34             L(j + 1:n, j) = (A(j + 1:n, j) - L(j + 1:n, 1:j - 1) * v(1:j - 1)') / v(j);
35         end;
36
37     else
38         v(1) = A(1, 1);
39         d(1) = v(1);
40         L(2:n, 1) = A(2:n, 1) / v(1);
41     end;
42
43 end;
44
45 % Put d into a matrix.
46 D = diag(d);
47 % Put ones on the diagonal of L.
48 L = L + eye(n);
49 U = D * L';
50 x = zeros(n, 1);
51 y = zeros(n, 1);
52
53 % calculate answers for Ly = b
54 for i = 1:1:n
55     alpha = 0;
56
57     for k = 1:1:i
58         alpha = alpha + L(i, k) * y(k);
59     end
60
61     y(i) = b(i) - alpha;
62 end
63
64 % calculate answers for Ux = y
65 for i = n:-1:1
66     alpha = 0;
67
68     for k = i + 1:1:n
69         alpha = alpha + U(i, k) * x(k);
70     end
71
72     x(i) = (y(i) - alpha) / U(i, i);
73 end
74
75 end

```

Per verificare che soluzioni sono giusti è stata creata la funzione testMatrixSolution.m che moltiplica valori di soluzione con la riga e somma tutti valori ottenuti dopodiché stampa questo valore, che deve essere uguale allo valore di stessa riga di b .

```

1 >> matrixDimension = 5;
2 >> A = round(10 * rand(matrixDimension));
3 >> A = A * A'
4 A =

```



```

5
6     170    189    102    166    90
7     189    278    96    203    115
8     102    96    93    108    41
9     166    203    108    195    103
10    90    115    41    103    87
11
12 >> b = round(10 * rand(matrixDimension, 1))
13 b =
14
15     2
16     6
17     3
18     1
19     4
20
21 >> solutions = es9_ldl(A, b)
22 solutions =
23
24    -0.2009
25     0.1788
26     0.3425
27    -0.3300
28     0.2468
29
30 >> testMatrixSolution(A,solutions)
31 2.0000
32 6.0000
33 3.0000
34 1
35 4.0000
36 >> ;
37 >> ;
38 >> ;
39 >> matrixDimension = 2;
40 >> A = round(10 * rand(matrixDimension));
41 >> A = A * A'
42 A =
43
44     162     81
45     81     65
46
47 >> b = round(10 * rand(matrixDimension, 1))
48 b =
49
50     2
51     9
52
53 >> solutions = es9_ldl(A, b)
54 solutions =
55
56    -0.1509
57     0.3265
58
59 >> testMatrixSolution(A,solutions)
60 2
61 9
62 >>

```

3.3 Esercizio 10

Data la function Matlab

```
1 function [A, b] = linsis(n, k, simme)
2     %
3     % [A,b] = linsis(n,k,simme) Crea una matrice A nxn ed un termine noto b,
4     % in modo che la soluzione del sistema lineare
5     % A*x=b sia x = [1,2,...,n]'.
6     % k `e un parametro ausiliario.
7     % simme, se specificato, crea una matrice
8     % simmetrica e definita positiva.
9     %
10    sigma = 10^(-2 * (1 - k)) / n;
11    rng(0);
12    [q1, r1] = qr(rand(n));
13
14    if nargin == 3
15        q2 = q1';
16    else
17        [q2, r1] = qr(rand(n));
18    end
19
20    A = q1 * diag([sigma 2 / n:1 / n:1]) * q2;
21    x = [1:n]';
22    b = A * x;
23    return
```

che crea sistemi lineari casuali con soluzione nota, risolvere, utilizzando la *function mialu*, i sistemi lineari generati da $[A, b] = \text{linsis}(10, 1)$ e $[A, b] = \text{linsis}(10, 10)$. Commentare l'accuratezza dei risultati ottenuti, dandone spiegazione esaustiva.

Soluzione:

3.4 Esercizio 11

Risolvere, utilizzando la *function mialdlt*, i sistemi lineari generati da $[A, b] = \text{linsis}(10, 1, 1)$ e $[A, b] = \text{linsis}(10, 10, 1)$. Commentare l'accuratezza dei risultati ottenuti, dandone spiegazione esaustiva.

Soluzione:

3.5 Esercizio 12

Scrivere una function Matlab,

```
1 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:

```
1 function [x, nr] = miaqr(A, b)
2     %   QR = myqr(A)
3     %   calcola la fattorizzazione QR di Householder della matrice A
4     %   Input:
5     %       A= matrice quadrata da fattorizzare
6     %
```

```

7 % Output:
8 %         QR=matrice contenente le informazioni sui fattori Q e R della
9 %         fattorizzazione QR di A
10 %
11 [m, n] = size(A);
12
13 if n > m
14     error('Dimensioni errate');
15 end
16
17 if length(b) ~= m
18     error('Dati inconsistenti');
19 end
20
21 QR = A;
22
23 for i = 1:n
24     alfa = norm(QR(i:m, i));
25
26     if alfa == 0
27         error('la matrice non ha rango massimo');
28     end
29
30     if QR(i, i) >= 0
31         alfa = -alfa;
32     end
33
34     v1 = QR(i, i) - alfa;
35     QR(i, i) = alfa;
36     QR(i + 1:m, i) = QR(i + 1:m, i) / v1;
37     beta = -v1 / alfa;
38     v = [1; QR(i + 1:m, i)];
39     QR(i:m, i + 1:n) = QR(i:m, i + 1:n) - (beta * v) * (v' * QR(i:m, i + 1:n));
40 end
41
42 [m, n] = size(QR);
43 k = length(b);
44
45 if k ~= m
46     error('Dati inconsistenti');
47 end
48
49 x = b(:);
50
51 for i = 1:n
52     v = [1; QR(i + 1:m, i)];
53     beta = 2 / (v' * v);
54     x(i:m) = x(i:m) - beta * (v' * x(i:m)) * v;
55 end
56
57 x = x(1:n);
58
59 for j = n:-1:1
60
61     if QR(j, j) == 0
62         error('Matrice singolare');
63     end
64

```

```

65         x(j) = x(j) / QR(j, j);
66         x(1:j - 1) = x(1:j - 1) - QR(1:j - 1, j) * x(j);
67     end
68
69 end
70
71 %{
72
73 A = round (10 * rand (3))
74 b = round (10 * rand (3, 1))
75
76 %}

```

3.6 Esercizio 13

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

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

definiti dai seguenti dati:

```

1 A = [ 1 3 2; 3 5 4; 5 7 6; 3 6 4; 1 4 2 ];
2 b = [ 15 28 41 33 22 ]';
3 D = diag(1:5);

```

Commentare i risultati ottenuti.

Soluzione:

```

1 %Codice esercizio 13
2
3 A= [1, 2, 3; 1 2 4; 3 4 5; 3 4 6; 5 6 7];
4 b=[14 17 26 29 38];
5 QR=myqr(A);
6 ris=qr.solve(QR,b);
7 disp(ris);

```

Il risultato finale è $ris = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$

3.7 Esercizio 14

Scrivere una function Matlab,

```

1 [x,nit] = newton(fun, jacobian, x0, tol, maxit)

```

che implementi efficientemente il metodo di Newton per risolvere sistemi di equazioni nonlineari. 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:

4 Capitolo 4

4.1 Esercizio 15

Usare la *function* del precedente esercizio per risolvere, a partire dal vettore iniziale nullo, i seguenti sistemi nonlineari, utilizzando tolleranze $tol = 1e-3, 1e-8, 1e-13$:

$$f_1(x) = \begin{pmatrix} (x_1^2 + 1)(x_2 - 2) \\ \exp(x_1 - 1) + \exp(x_2 - 2) - 2 \end{pmatrix}, \quad f_2(x) = \begin{pmatrix} x_1 - x_2 x_3 \\ \exp(x_1 + x_2 + x_3 - 3) - x_2 \\ x_1 + x_2 + 2x_3 - 4 \end{pmatrix}.$$

Tabulare i risultati ottenuti, commentandone l'accuratezza.

Soluzione:

4.2 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:

4.3 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:

4.4 Esercizio 18

Costruire una function, *hermite.m*, avente una sintassi analoga alla function *spline* di Matlab (in pratica, con un parametro di ingresso in più per i valori delle derivate), che implementi, in modo vettoriale, il polinomio interpolante di Hermite.

Soluzione:

4.5 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:

4.6 Esercizio 20

Costruire una function, *spline0.m*, avente la stessa sintassi della function *spline* di Matlab, che implementi la spline cubica naturale interpolante una funzione.

Soluzione:

5 Capitolo 5

5.1 Esercizio 21

Costruire una tabella in cui viene riportato, al crescere di n , il massimo errore di interpolazione ottenuto approssimando la funzione:

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

sulle ascisse $x_0 \leq x_1 \leq \dots \leq x_n$:

- equidistanti in $[-2, 3]$,
- di Chebyshev per lo stesso intervallo,

utilizzando le function degli Esercizi 16-18 e 20, e la function *spline* di Matlab. Considerare $n = 4, 8, 16, \dots, 40$ e stimare l'errore di interpolazione su 10001 punti equidistanti nell'intervallo $[x_0, x_n]$.

Soluzione:

5.2 Esercizio 22

Tabulare il massimo errore di approssimazione (stimato su 10001 punti equidistanti in $[0, 1]$) ottenuto approssimando le funzioni

$$\sin(2\pi x) \quad e \quad \cos(2\pi x)$$

mediante le function *spline0* e *spline*, interpolanti su $n+1$ punti equidistanti in $[0, 1]$, per $n = 5, 10, 15, 20, \dots, 50$. Commentare i risultati ottenuti.

Soluzione:

5.3 Esercizio 23

Sia assegnata la seguente perturbazione della funzione $f(x) = \sin(\pi x^2)$:

$$\tilde{f}(x) = f(x) + 10^{-1} \text{rand}(\text{size}(x)),$$

in cui *rand* è la function built-in di Matlab. Calcolare polinomio di approssimazione ai minimi quadrati di grado m , $p(x)$, sui dati $(x_i, \tilde{f}(x_i))$, $i = 0, \dots, n$, con:

$$x_i = i/n, \quad n = 10^4.$$

Graficare (in formato *semilogy*) l'errore di approssimazione $\|f - p\|$ (stimato come il massimo errore sui punti x_i), relativo all'intervallo $[0, 1]$, rispetto ad m , per $m = 1, 2, \dots, 15$. Commentare i risultati ottenuti.

Soluzione:

5.4 Esercizio 24

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, \dots , 7 e 9 (come numeri razionali).

Soluzione:

5.5 Esercizio 25

Utilizzare le formule tabulate nel precedente esercizio per calcolare le approssimazioni dell'integrale

$$I(f) = \int_0^1 e^{3x} dx,$$

tabulando (in modo significativo) il corrispondente errore di quadratura (risolvere a mano l'integrale).

Soluzione:

5.6 Esercizio 26

Scrivere una function Matlab,

$$[If, err, nfeval] = composita(fun, a, b, n, tol)$$

in cui

- fun è l'identificatore di una function che calcoli (in modo vettoriale) la funzione integranda,
- a e b sono gli estremi dell'intervallo di integrazione,
- n è il grado di una formula di Newton-Cotes base,
- tol è l'accuratezza richiesta,

che calcoli, fornendo la stima err dell'errore di quadratura, l'approssimazione If dell'integrale, raddoppiando il numero di punti ed usando la formula composita corrispondente per stimare l'errore di quadratura, fino a soddisfare il requisito di accuratezza richiesto. In uscita è anche il numero totale di valutazioni funzionali effettuate, $nfeval$.

N.B.: evitare di effettuare valutazioni di funzione ridondanti.

Soluzione:

5.7 Esercizio 30

Tabulare il numero di valutazioni di funzione richieste per calcolare, mediante la function del precedente esercizio, l'approssimazione dell'integrale

$$I(f) = \int_0^1 \sin\left(\frac{1}{0.1+x}\right) dx,$$

utilizzando le formule di Newton-Cotes di grado $n = 1, \dots, 7$, e 9 , e tolleranze $tol = 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}, 10^{-6}$.

Soluzione:

5.8 Esercizio 28

Scrivere una function che implementi la formula adattiva dei trapezi.

N.B.: evitare di effettuare valutazioni di funzione ridondanti.

Soluzione:

5.9 Esercizio 29

Scrivere una function che implementi la formula adattiva di Simpson.

N.B.: evitare di effettuare valutazioni di funzione ridondanti.

Soluzione:

5.10 Esercizio 30

Tabulare il numero di valutazioni di funzione richieste dalle function degli Esercizi 29 e 30 per approssimare l'integral

$$I(f) = \int_0^1 \sin\left(\frac{1}{0.01+x}\right) dx,$$

con tolleranze $tol = 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}, 10^{-6}$.

Soluzione:

6 Codici ausiliari

6.1 Esercizio 6

Listing 1: es6.m

```
1 syms x;
2 f = @(x)(x - cos((pi / 2) * x));
3 f1 = matlabFunction(diff(f, x));
4
5 x0 = 1;
6 x1 = 9.9e-1;
7 x = zeros(4, 3);
8 y = zeros(4, 3);
9 colnames = {};
10
11 for i = 3:3:12
12     colnames{end + 1} = ['10^-' num2str(i)];
13     [x(i / 3, 1), y(i / 3, 1)] = es5_newton(f, f1, x0, 10^(-i));
14     [x(i / 3, 2), y(i / 3, 2)] = es5_secanti(f, x0, x1, 10^(-i));
15     [x(i / 3, 3), y(i / 3, 3)] = es5_steffensen(f, x0, 10^(-i));
16 end
17
18 row_names = {'newton', 'secanti', 'steffensen'};
19 disp(x)
20 figure
21 plot([3, 6, 9, 12], y, 'o-')
22 title('iterazioni richieste per la convergenza al diminuire di tol x')
23 xlabel('tolleranza = 10^{-x}')
24 ylabel('iterazioni')
25 legend(row_names, 'Location', 'northwest')
```

6.2 Esercizio 7

Listing 2: es7.m

```
1 syms x;
2 f = @(x)((x - cos((pi / 2) * x))^3);
3 f1 = matlabFunction(diff(f, x));
4
5 x0 = 1;
6 x1 = 9.9e-1;
7 x = zeros(4, 3);
8 y = zeros(4, 3);
9 colnames = {};
10
11 for i = 3:3:12
12     colnames{end + 1} = ['10^-' num2str(i)];
13     [x(i / 3, 1), y(i / 3, 1)] = es5_newton(f, f1, x0, 10^(-i));
14     [x(i / 3, 2), y(i / 3, 2)] = es5_secanti(f, x0, x1, 10^(-i));
15     [x(i / 3, 3), y(i / 3, 3)] = es5_steffensen(f, x0, 10^(-i));
16 end
17
18 row_names = {'newton', 'secanti', 'steffensen'};
19 disp(x)
20 figure
21 plot([3, 6, 9, 12], y, 'o-')
22 title('iterazioni richieste per la convergenza al diminuire di tol x')
```



```

23 xlabel('tolleranza = 10^{-x}')
24 ylabel('iterazioni')
25 legend(row_names, 'Location', 'northwest')

```

6.3 Esercizi 8 e 9

Listing 3: testMatrixSolution.m

```

1 function testMatrixSolution(A, solution)
2     % Syntax: testMatrixSolution(A, solution)
3     % The function tests calculates result of A with solution as variables
4     % Input:  A      – a matrix representation of the equations with unknown variables
5     % Output: prints the result of A with solutions as variables
6     matrixDimension = size(A)(1);
7
8     for j = 1:1:matrixDimension;
9         total = 0;
10
11         for i = 1:1:matrixDimension;
12             total = total + (A(j, i) * solution(i));
13         end;
14
15         disp(total);
16     end
17
18 end

```

6.4 Esercizio 15

Listing 4: es15.m

```

1 %Codice esercizio 15
2
3 f = @(x)(cos((pi*x.^2)/2));
4 x = linspace(-1, 1, 100001);
5 linerrors = zeros(1, 40);
6 chebyerrors = zeros(1, 40);
7 for n = 1:40
8     xlin = linspace(-1, 1, n+1);
9     xcheby = chebyshev(-1,1,n+1);
10    ylin = lagrange(xlin,f(xlin),x);
11    ycheby = lagrange(xcheby,f(xcheby),x);
12    linerrors(n) = norm(abs(f(x) - ylin), inf);
13    chebyerrors(n) = norm( abs(f(x) - ycheby), inf);
14 end
15 semilogy(linerrors);
16 hold on;
17 semilogy(chebyerrors);
18 xlabel('numero di ascisse di interpolazione');
19 ylabel('massimo errore di interpolazione');
20 legend({'ascisse equidistanti', 'ascisse di chebyshev'}, 'Location', 'northeast');

```

6.5 Esercizio 16

Listing 5: es16.m

```

1 %Codice esercizio 16
2
3 f = @(x)(cos((pi*x.^2)/2));
4 f1 = @(x)(-pi*x.*sin((pi*x.^2)/2));
5 x = linspace(-1, 1, 100001);
6 linerrors = zeros(1, 20);
7 chebyerrors = zeros(1, 20);
8 for n = 1:20
9     xlin = linspace(-1, 1, n+1);
10    xcheby = chebyshev(-1,1, n+1);
11    ylin = hermite(xlin,f(xlin),f1(xlin),x);
12    ycheby = hermite(xcheby,f(xcheby),f1(xcheby),x);
13    linerrors(n) = norm(abs(f(x) - ylin), inf);
14    chebyerrors(n) = norm( abs(f(x) - ycheby), inf);
15 end
16 semilogy(linerrors);
17 hold on;
18 semilogy(chebyerrors);
19 xlabel('numero di ascisse di interpolazione');
20 ylabel('massimo errore di interpolazione');
21 legend({'ascisse equidistanti', 'ascisse di chebyshev'}, 'Location', 'northeast');

```

6.6 Esercizio 18

Listing 6: es18.m

```

1 %Codice esercizio 18
2 f = @(x)(cos((pi*(x.^2))/2));
3 x = linspace(-1, 1, 100001);
4 linerrors = zeros(1, 40);
5 chebyerrors = zeros(1, 40);
6 for n = 4:100
7     xlin = linspace(-1, 1, n+1);
8     xcheby = chebyshev(-1,1,n+1);
9     %xcheby(1)=-1;
10    %xcheby(n+1)=1;
11    ylin = splinenat(xlin,f(xlin),x);
12    ycheby = splinenat(xcheby,f(xcheby),x);
13    ylin=ylin';
14    ycheby=ycheby';
15    linerrors(n) = norm(abs(f(x) - ylin), inf);
16    chebyerrors(n) = norm( abs(f(x) - ycheby), inf);
17 end
18 semilogy(linerrors);
19 hold on;
20 semilogy(chebyerrors);
21 xlabel('numero di ascisse di interpolazione');
22 ylabel('massimo errore di interpolazione');
23 legend({'ascisse equidistanti', 'ascisse di chebyshev'}, 'Location', 'northeast');

```

6.7 Esercizio 19

Listing 7: es19.m

```

1 %Codice esercizio 19
2 f = @(x)(cos((pi*(x.^2))/2));
3 x = linspace(-1, 1, 100001);

```

```

4 linerrors = zeros(1, 40);
5 chebyerrors = zeros(1, 40);
6 for n = 4:100
7     xlin = linspace(-1, 1, n+1);
8     xcheby = chebyshev(-1,1,n+1);
9     ylin = spline(xlin,f(xlin),x);
10    ycheby = spline(xcheby,f(xcheby),x);
11    linerrors(n) = norm(abs(f(x) - ylin), inf);
12    chebyerrors(n) = norm( abs(f(x) - ycheby), inf);
13 end
14 semilogy(linerrors);
15 hold on;
16 semilogy(chebyerrors);
17 xlabel('numero di ascisse di interpolazione');
18 ylabel('massimo errore di interpolazione');
19 legend({'ascisse equidistanti', 'ascisse di chebyshev'}, 'Location', 'northeast');

```

6.8 Esercizio 20

Listing 8: es20.m

```

1 %Codice esercizio 20
2
3 f = @(x)(cos((pi*x.^2)/2));
4 fp = @(x)(f(x) + 10^(-3)*rand(size(x)));
5 xi = -1 + 2*(0:10^4)/10^4;
6 fi = f(xi);
7 fpi = fp(xi);
8 errors=zeros(1, 20);
9 for m = 1:20
10    y = minimiquadrati(xi, fpi, m);
11    errors(m) = norm(abs(y-fi), inf);
12 end
13 semilogy(errors);
14 xlabel('grado del polinomio');
15 ylabel('errore di interpolazione massimo');

```

6.9 Esercizio 21

Listing 9: es21.m

```

1 %Codice esercizio 21
2
3 for i = 1:7
4     weights= rats(ncweights(i))
5 end

```

6.10 Esercizio 22

Listing 10: es22.m

```

1 %Codice esercizio 22
2
3 rapp = zeros(1, 50);
4 for i = 1:50
5     rapp(i) = sum(abs(ncweights(i)))/i;

```

```

6 end
7 semilogy(rapp);
8 xlabel('grado n della formula di Newton-Cotes');
9 ylabel('^{\mathcal{K}_n}/_{\mathcal{K}}');

```

6.11 Esercizio 23

Listing 11: es23.m

```

1 %Codice esercizio 23
2
3 value = log(cos(1)/cos(1.1));
4 x = zeros(1,9);
5 errors=zeros(1, 9);
6 for i = 1:9
7     x(i) = newtoncotes(@tan, -1,1.1, i);
8     errors(i) = abs(value-x(i));
9 end

```

6.12 Esercizio 24

Listing 12: es24.m

```

1 %Codice esercizio 24
2
3 a = -1;
4 b = 1.1;
5 n = 10;
6 itrap = zeros(1, n);
7 isimp = zeros(1, n);
8 for i = 1:n
9     itrap(i) = trapecomp(@tan, a, b, i*2);
10    isimp(i) = simpcomp(@tan, a, b, i*2);
11 end
12 integrali = [itrap; isimp];
13 row_names = {'trapezi composta', 'simpson composta'};
14 colnames = {'2', '4', '6', '8', '10', '12', '14', '16', '18', '20'};
15 values = array2table(integrali, 'RowNames', row_names, 'VariableNames', colnames);
16 disp(values);

```

6.13 Esercizio 25

Listing 13: es25.m

```

1 %Codice esercizio 25
2
3 format long e
4 f = @(x)(1/(1+100*x.^2));
5 a = -1;
6 b = 1;
7 itrap = zeros(1, 5);
8 trap_points = zeros(1, 5);
9 isimp = zeros(1, 5);
10 simp_points = zeros(1, 5);
11 for i = 1:5
12     [itrap(i), points] = adaptrap(f, a, b, 10^(-i-1));

```

```

13     trap_points(i) = length(points);
14     [isimp(i), points] = adapsim(f, a, b, 10^(-i-1));
15     simp_points(i) = length(points);
16 end
17 integrali = [itrap; isimp];
18 npoints = [trap_points; simp_points];
19 row_names = {'trapezi adattiva', 'simpson adattiva'};
20 colnames = {'10^-2', '10^-3', '10^-4', '10^-5', '10^-6'};
21 values = array2table(integrali, 'RowNames', row_names, 'VariableNames', colnames);
22 npoints = array2table(npoints, 'RowNames', row_names, 'VariableNames', colnames);
23 disp(values);
24 format
25 disp(npoints);

```