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 2, 2022

Contents

List of Figures

List of Tables

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 cosi:

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

Calcoliamo le sviluppi di Taylor per tutti funzioni che sono presenti in formula da verificare. A variablie 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:

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

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

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

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 $\mathbf{M}$ é finito. Pertanto é 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 é l'errore relativo della funzione tale che $\varepsilon_x \leq u$,

$$u=\begin{cases} 2^{1-m}, & pertroncamento \\ \frac{1}{2}*2^{1-m}, & perarrotondamento \end{cases}$$

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:

```
>> a=eps
a = 2.2204e-16
>> truncated = 2^(1-53)
truncated = 2.2204e-16
>> rounded = (1/2)*(2^(1-53))
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:

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

Spiegare i risultati ottenuti.
Soluzione:

```
>> format long e
>> (1+(1e-14-1))*1e14
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 é 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 'e ottenuta la massima precisione possibile.
Soluzione:

```
function answer = radice(x)
    % answer = radice(x)
    % This method is called Babilons method
    % It gets some value in input and apply the Babilons method
    % until I needed precision, but we need maximum precision possible,
    % so I apply this method until I get two equal values
    % one before another and this will mean this method arrived at a point,
    % in which value not changes, so it is a maximum precision.
    % INPUT:  x      - Is a value of which we want to get square root
    % OUTPUT: answer - This is the square root of the input
    format long e;

    if x < 0
        error('Wrong value in input! Should be not less then zero!');
    end

    answer = x / 2;
    guess = x;

    while not(answer == guess)
        guess = answer;
        answer = (guess + (x / guess)) / 2;
    end

end

for i = 1:0.5:10
    s = 1e-10 + 1e10 * log10(i);
    custom = radice(s);
    native = sqrt(s);

    if abs(custom - native) >= eps(abs(native))

        if abs(custom - native) > eps(abs(native))
            diff = abs(custom - native);
            fprintf([ ...
                    'result is not equal for %.' int2str(abs(int16(log10(eps(s)))) + 1) ...
                    'd, with values custom=%.' int2str(abs(int16(log10(eps(custom)))) + 20) ...
                    'd and native=%.' int2str(abs(int16(log10(eps(native)))) + 20) ...
                    'd, with diff=%.' int2str(abs(int16(log10(eps(diff)))) + 20) 'd\n' ...
                ], s, custom, native, diff);
        elseif abs(custom - native) == eps(abs(native))
            fprintf([ ...
                    'Difference betwen this numbers are equal to exponent ' ...
                    'for this numbers, and equals to %.30d\n'...
```

```
                ], eps(abs(native)));
        end

    end

    fprintf([ ...
            'custom=%.' int2str(abs(int16(log10(eps(custom)))) + 1) ...
            'd\nnative=%.' int2str(abs(int16(log10(eps(native)))) + 1) ...
            'd\n\n'
        ], custom, native);

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:

$$x_{n+1} = x_n - \frac{f(x_n)^2}{f(x_n + f(x_n)) - f(x_n)}, \qquad \text{n=0,1,...}$$

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

```
function [x, i] = newton(f, f1, x0, tolx, maxit)
    % [x, i] = newton(f, f1, x0, tolx, maxit)
    % Newton's method, used to calculate a root of the equation f(x)=0
    % Input:  f     — function that we can derive
    %         f1    — derived function of the 'f'
    %         x0    — initial approximation
    %         tolx  — tolerance
    %         maxit — maximum number of iterations (default = 100)
    % Output: x     — the approximation of the root function
    %         i     — number of iterations
    % CHECK THIS: bisezione, corde, secanti, aitken, newtonmod

    format long e;

    if nargin < 4
        error('Number of arguments must be at least 4!');
    elseif nargin == 4
        maxit = 100;
    end

    if tolx < eps
        error('Tolerance cannot be checked!');
```

```
    end

    precisionEnough = false;

    for i = 1:maxit
        fx = feval(f, x0);
        f1x = feval(f1, x0);

        if f1x == 0
            warning(['Value of f1 on iteration ' int2str(i) ' is zero!']);
            break;
        end

        x = x0 - (fx / f1x);

        precisionEnough = abs(x - x0) <= tolx * (1 + abs(x0));

        if precisionEnough
            break;
        end

        x0 = x;

    end

    if ~precisionEnough
        warning(['Failed to converge in ' maxit ' iterations!']);
    end

end
```

- Metodo delle secanti

```
function [x, i] = secanti(f, x0, x1, tolx, maxit)
    % [x, i] = secanti(f, x0, x1, tolx, maxit)
    % Method of the secant, used to calculate a root of the equation f(x)=0
    % Input:  f     - function that we can derive
    %         x0    - initial approximation
    %         x1    - second initial approximation
    %         tolx  - tolerance
    %         maxit - maximum number of iterations (default = 100)
    % Output: x     - the approximation of the root function
    %         i     - number of iterations
    % CHECK THIS: bisezione, corde, secanti, aitken, newtonmod

    format long e;

    if nargin < 4
        error('Number of arguments must be at least 4!');
    elseif nargin == 4
        maxit = 100;
    end

    precisionEnough = false;
    i = 0;
    f0 = feval(f, x0);

    for i = 1:maxit
        f1 = feval(f, x1);
```

```
        df1 = (f1 - f0) / (x1 - x0);

        if df1 == 0
            warning(['Delta of f on iteration ' int2str(i) ' is zero!']);
            break;
        end

        x0 = x1;
        x1 = x1 - (f1 / df1);
        f0 = f1;
        precisionEnough = abs(x1 - x0) <= tolx * (1 + abs(x0));

        if precisionEnough
            break;
        end

    end

    if ~precisionEnough
        warning(['Failed to converge in ' maxit ' iterations!']);
    end

    x = x1;
end
```

- Metodo di Steffensen

```
function [x, i] = steffensen(f, x0, tolx, maxit)
    % [x, i] = steffensen(f, x0, tolx, maxit)
    % Steffensen's method, used to calculate a root of the equation f(x)=0
    % Input:  f     - a fixed point iteration function
    %         x0    - initial guess to the fixed point
    %         tolx  - tolerance
    %         maxit - maximum number of iterations (default = 100)
    % Output: x     - the approximation of the root function
    %         i     - number of iterations
    % CHECK THIS: bisezione, corde, secanti, aitken, newtonmod

    format long e;

    if nargin < 3
        error('Number of arguments must be at least 3!');
    elseif nargin == 3
        maxit = 100;
    end

    precisionEnough = false;

    for i = 1:maxit
        % get ready to do a large, but finite, number of iterations.
        % This is so that if the method fails to converge, we won't
        % be stuck in an infinite loop.
        fx = feval(f, x0); % calculate the next two guesses for the fixed point.
        f_fx_x = feval(f, fx + x0);

        if (f_fx_x - fx) == 0
            warning(['Distance betwen two guesses on iteration ' int2str(i) ' is
                zero!']);
            break;
```

```
        end

        x = x0 - (fx^2 / (f_fx_x - fx));
        % use Aitken's delta squared method to
        % find a better approximation to x0.
        precisionEnough = abs(x - x0) <= tolx * (1 + abs(x0));

        if precisionEnough
            break; % if we are, stop the iterations, we have our answer.
        end

        x0 = x;

    end

    if ~precisionEnough
        warning(['Failed to converge in ' int2str(maxit) ' iterations!']);
    end

end
```

2.3 Esercizio 6

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

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

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: $2i$, $i = 1 : maxit$;

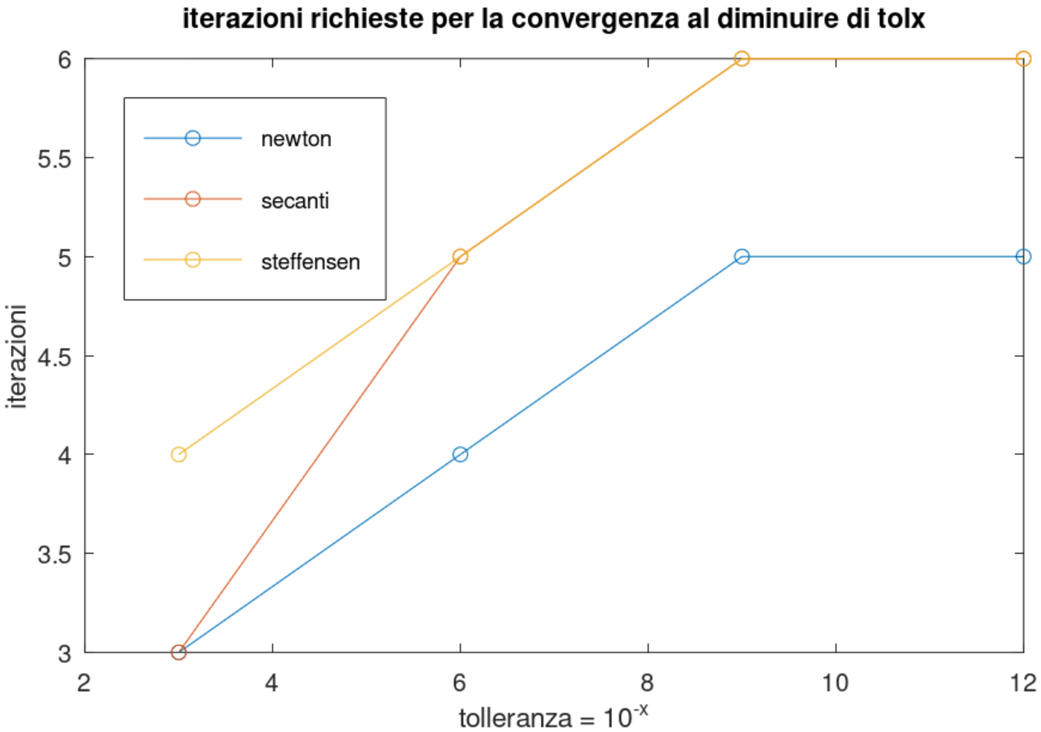

Figure 1: iterazioni richieste

2.4 Esercizio 7

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

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

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.msi 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 falliscie 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 divizione a zero.

Costo computazionale

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

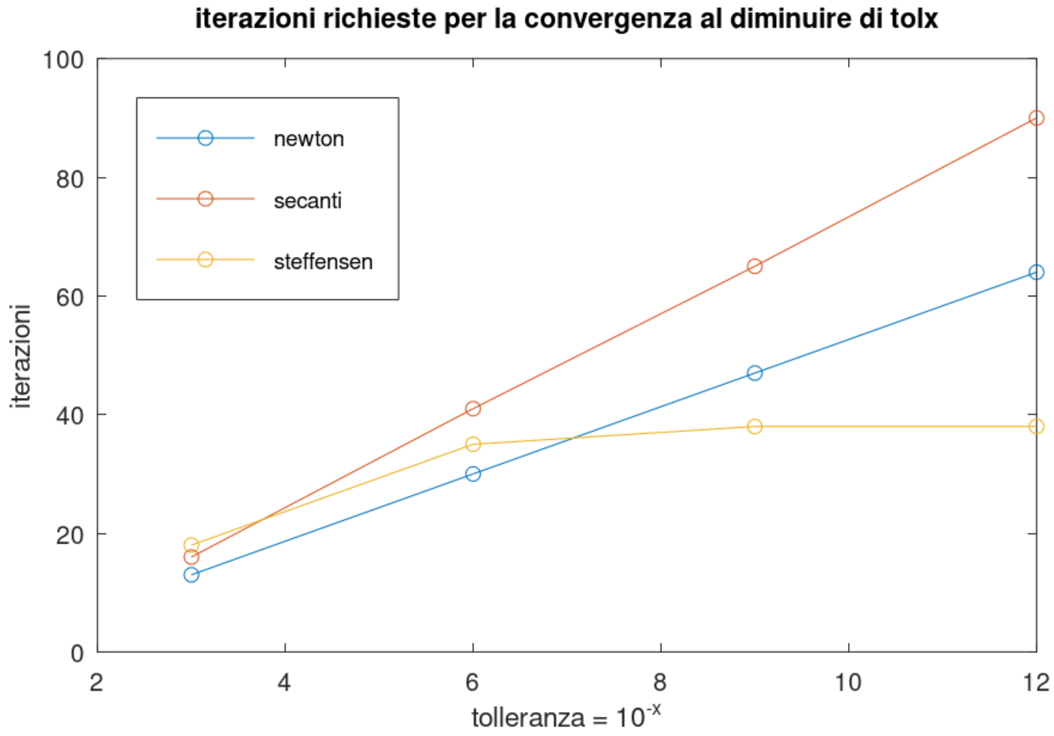

Figure 2: iterazioni richieste

- 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,

```
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:

```
function x = mialu(A, b)
    % x = mialu(A, b)
    % Method of the secant, used to calculate a root of the equation f(x)=0
    % Input:  A     - a matrix rapresentation of the equations with unknown variables
    %         b     - result of unknown variables
    % Output: x     - array to what equals unknown variables

    % Calculate L and U matrixes
    n = length(A);
    L = zeros(n);
    U = zeros(n);
    % P = eye(n);

    for k = 1:n
        % find the entry in the left column with the largest abs value (pivot)
        [~, r] = max(abs(A(k:end, k)));
        r = n - (n - k + 1) + r;

        A([k r], :) = A([r k], :);
        % P([k r], :) = P([r k], :);
        L([k r], :) = L([r k], :);
        b([k r], :) = b([r k], :);

        % from the pivot down divide by the pivot
        L(k:n, k) = A(k:n, k) / A(k, k);

        U(k, 1:n) = A(k, 1:n);
        A(k + 1:n, 1:n) = A(k + 1:n, 1:n) - L(k + 1:n, k) * A(k, 1:n);

    end

    U(:, end) = A(:, end);

    x = zeros(n, 1);
    y = zeros(n, 1);

    % calcolo delle soluzioni di Ly = b
    for i = 1:1:n
        alpha = 0;

        for k = 1:1:i
            alpha = alpha + L(i, k) * y(k);
        end

        y(i) = b(i) - alpha;
    end
```

```

    % calcolo delle soluzioni di Ux = y
    for i = n:-1:1
        alpha = 0;

        for k = i + 1:1:n
            alpha = alpha + U(i, k) * x(k);
        end

        x(i) = (y(i) - alpha) / U(i, i);
    end

end
```

3.2 Esercizio 9

Scrivere una function Matlab,

```
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:

```
function x = mialdl(A, b)
    % x = mialdl(A, b)
    % Method of the secant, used to calculate a root of the equation f(x)=0
    % Input:  A     - a matrix rapresentation of the equations with unknown variables
    %                 It is assumed that A is symmetric and postive definite.
    %         b     - result of unknown variables
    % Output: x     - array to what equals unknown variables

    %  Figure out the size of A.
    n = size(A, 1);
    % The main loop.  See Golub and Van Loan for details.
    L = zeros(n, n);

    for j = 1:n,

        if (j > 1),
            v(1:j - 1) = L(j, 1:j - 1) .* d(1:j - 1);
            v(j) = A(j, j) - L(j, 1:j - 1) * v(1:j - 1)';
            d(j) = v(j);

            if (j < n),
                L(j + 1:n, j) = (A(j + 1:n, j) - L(j + 1:n, 1:j - 1) * v(1:j - 1)') / v(j
                    );
            end;

        else
            v(1) = A(1, 1);
            d(1) = v(1);
            L(2:n, 1) = A(2:n, 1) / v(1);
        end;

    end;

```

```
    %  Put d into a matrix.
    D = diag(d);
    %  Put ones on the diagonal of L.
    L = L + eye(n);

    % n = size(A,1);
    % L=zeros(n,n);
    % for j=1:n,
    %     if (j>1),
    %         v(1:j-1)=L(j,1:j-1).*d(1:j-1);
    %         v(j)=A(j,j)-L(j,1:j-1)*v(1:j-1)';
    %         d(j)=v(j);
    %         if(j<n),
    %             L(j+1:n,j)=(A(j+1:n,j)-L(j+1:n,1:j-1)*v(1:j-1)')/v(j);
    %         end;
    %     else
    %         v(1)=A(1,1);
    %         d(1)=v(1);
    %         L(2:n,1)=A(2:n,1)/v(1);
    %     end;
    % end;
    % D=diag(d);
    % L=L+eye(n);

    U = D * L';
    x = zeros(n, 1);
    y = zeros(n, 1);

    % calcolo delle soluzioni di Ly = b
    for i = 1:1:n
        alpha = 0;

        for k = 1:1:i
            alpha = alpha + L(i, k) * y(k);
        end

        y(i) = b(i) - alpha;
    end

    % calcolo delle soluzioni di Ux = y
    for i = n:-1:1
        alpha = 0;

        for k = i + 1:1:n
            alpha = alpha + U(i, k) * x(k);
        end

        x(i) = (y(i) - alpha) / U(i, i);
    end

end
```

3.3 Esercizio 10

Data la function Matlab

```
function [A, b] = linsis(n, k, simme)
    %
```

```
    % [A,b] = linsis(n,k,simme) Crea una matrice A nxn ed un termine noto b,
    % in modo che la soluzione del sistema lineare
    % A*x=b sia x = [1,2,...,n]'.
    % k `e un parametro ausiliario.
    % simme, se specificato, crea una matrice
    % simmetrica e definita positiva.
    %
    sigma = 10^(−2 * (1 − k)) / n;
    rng(0);
    [q1, r1] = qr(rand(n));

    if nargin == 3
        q2 = q1';
    else
        [q2, r1] = qr(rand(n));
    end

    A = q1 * diag([sigma 2 / n:1 / n:1]) * q2;
    x = [1:n]';
    b = A * x;
    return
```

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

3.4 Esercizio 11

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

3.5 Esercizio 12

Scrivere una function Matlab,

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

che, data in ingresso la matrice A $m \times n$, con $m \geq n = rank(A)$, 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:

```
function [x, nr] = miaqr(A, b)
    %   QR = myqr(A)
    %   calcola la fattorizzazione QR di Householder della matrice A
    %   Input:
    %           A= matrice quadrata da fattorizzare
    %
    %   Output:
    %           QR=matrice contenente le informazioni sui fattori Q e R della
    %           fattorizzazione QR di A
    %
    [m, n] = size(A);
```

```

    if n > m
        error('Dimensioni errate');
    end

    if length(b) ~= m
        error('Dati inconsistenti');
    end

    QR = A;

    for i = 1:n
        alfa = norm(QR(i:m, i));

        if alfa == 0
            error('la matrice non ha rango massimo');
        end

        if QR(i, i) >= 0
            alfa = -alfa;
        end

        v1 = QR(i, i) -alfa;
        QR(i, i) = alfa;
        QR(i + 1:m, i) = QR(i + 1:m, i) / v1;
        beta = -v1 / alfa;
        v = [1; QR(i + 1:m, i)];
        QR(i:m, i + 1:n) = QR(i:m, i + 1:n) - (beta * v) * (v' * QR(i:m, i + 1:n));
    end

    [m, n] = size(QR);
    k = length(b);

    if k ~= m
        error('Dati inconsistenti');
    end

    x = b(:);

    for i = 1:n
        v = [1; QR(i + 1:m, i)];
        beta = 2 / (v' * v);
        x(i:m) = x(i:m) - beta * (v' * x(i:m)) * v;
    end

    x = x(1:n);

    for j = n:-1:1

        if QR(j, j) == 0
            error('Matrice singolare');
        end

        x(j) = x(j) / QR(j, j);
        x(1:j - 1) = x(1:j - 1) - QR(1:j - 1, j) * x(j);
    end

end
```

```
%{

A = round (10 * rand (3))
b = round (10 * rand (3, 1))

%}
```

3.6 Esercizio 13

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

$$\mathrm{A\ x = b}, \qquad \mathrm{(D{*}A)x = (D{*}b)}$$

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 = diag(1:5);
```

Commentare i risultati ottenuti.
Soluzione:

```
%Codice esercizio 13

A= [1, 2, 3; 1 2 4; 3 4 5; 3 4 6; 5 6 7];
b=[14 17 26 29 38];
QR=myqr(A);
ris=qrsolve(QR,b);
disp(ris);
```

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

3.7 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 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 $tol = 1e-3$, $1e-8$, $1e-13$:

$$f_1(x) = \begin{pmatrix} (x_1^2+1)(x_2-2) \\ exp(x_1-1)+exp(x_2-2)-2 \end{pmatrix}, \qquad f_2(x) = \begin{pmatrix} x_1 - x_2x_3 \\ exp(x_1+x_2+x_3-3)-x_2 \\ x_1+x_2+2x_3-4 \end{pmatrix}.$$

Tabulare i risultati ottenuti, commentandone l'accuratezza.
Soluzione:

4.2 Esercizio 16

Costruire una function, *lagrange.m*, avente la <u>stessa sintassi</u> 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 <u>stessa sintassi</u> 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 <u>analoga</u> alla function *spline* di Matlab (in pratica, con un parametro di ingresso in pi‘u 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 <u>stessa sintassi</u> 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 $x_0 \leq x_1 \cdots \leq x_n$:

- 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 $n = 4, 8, 16, \ldots, 40$ e stimare l'errore di interpolazione su 10001 punti equidistanti nell'intervallo $[x0, xn]$.
Soluzione:

5.2 Esercizio 22

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

$$sin(2\Pi x) \qquad e \qquad cos(2\Pi x)$$

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

5.3 Esercizio 23

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

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

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

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

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 $m = 1, 2, \ldots, 15$. 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, ..., 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 qui

- 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, . . . , 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

```
syms x;
f = @(x)(x - cos((pi / 2) * x));
f1 = matlabFunction(diff(f, x));

x0 = 1;
x1 = 9.9e-1;
x = zeros(4, 3);
y = zeros(4, 3);
colnames = {};

for i = 3:3:12
    colnames{end + 1} = ['10^-' num2str(i)];
    [x(i / 3, 1), y(i / 3, 1)] = es5_newton(f, f1, x0, 10^(-i));
    [x(i / 3, 2), y(i / 3, 2)] = es5_secanti(f, x0, x1, 10^(-i));
    [x(i / 3, 3), y(i / 3, 3)] = es5_steffensen(f, x0, 10^(-i));
end

row_names = {'newton', 'secanti', 'steffensen'};
disp(x)
figure
plot([3, 6, 9, 12], y, 'o-')
title('iterazioni richieste per la convergenza al diminuire di tolx')
xlabel('tolleranza = 10^{-x}')
ylabel('iterazioni')
legend(row_names, 'Location', 'northwest')
```

6.2 Esercizio 7

Listing 2: es7.m

```
syms x;
f = @(x)((x - cos((pi / 2) * x))^3);
f1 = matlabFunction(diff(f, x));

x0 = 1;
x1 = 9.9e-1;
x = zeros(4, 3);
y = zeros(4, 3);
colnames = {};

for i = 3:3:12
    colnames{end + 1} = ['10^-' num2str(i)];
    [x(i / 3, 1), y(i / 3, 1)] = es5_newton(f, f1, x0, 10^(-i));
    [x(i / 3, 2), y(i / 3, 2)] = es5_secanti(f, x0, x1, 10^(-i));
    [x(i / 3, 3), y(i / 3, 3)] = es5_steffensen(f, x0, 10^(-i));
end

row_names = {'newton', 'secanti', 'steffensen'};
disp(x)
figure
plot([3, 6, 9, 12], y, 'o-')
title('iterazioni richieste per la convergenza al diminuire di tolx')
```

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

6.3 Esercizio 15

Listing 3: es15.m

```
%Codice esercizio 15

f = @(x)(cos((pi*x.^2)/2));
x = linspace(-1, 1, 100001);
linerrors = zeros(1, 40);
chebyerrors = zeros(1, 40);
for n = 1:40
    xlin = linspace(-1, 1, n+1);
    xcheby = chebyshev(-1,1,n+1);
    ylin = lagrange(xlin,f(xlin),x);
    ycheby = lagrange(xcheby,f(xcheby),x);
    linerrors(n) = norm(abs(f(x) - ylin), inf);
    chebyerrors(n) = norm( abs(f(x) - ycheby), inf);
end
semilogy(linerrors);
hold on;
semilogy(chebyerrors);
xlabel('numero di ascisse di interpolazione');
ylabel('massimo errore di interpolazione');
legend({'ascisse  equidistanti', 'ascisse di chebyshev'},'Location','northeast');
```

6.4 Esercizio 16

Listing 4: es16.m

```
%Codice esercizio 16

f = @(x)(cos((pi*x.^2)/2));
f1 = @(x)(-pi*x.*sin((pi*x.^2)/2));
x = linspace(-1, 1, 100001);
linerrors = zeros(1, 20);
chebyerrors = zeros(1, 20);
for n = 1:20
    xlin = linspace(-1, 1, n+1);
    xcheby = chebyshev(-1,1, n+1);
    ylin = hermite(xlin,f(xlin),f1(xlin),x);
    ycheby = hermite(xcheby,f(xcheby),f1(xcheby),x);
    linerrors(n) = norm(abs(f(x) - ylin), inf);
    chebyerrors(n) = norm( abs(f(x) - ycheby), inf);
end
semilogy(linerrors);
hold on;
semilogy(chebyerrors);
xlabel('numero di ascisse di interpolazione');
ylabel('massimo errore di interpolazione');
legend({'ascisse  equidistanti', 'ascisse di chebyshev'},'Location','northeast');
```

6.5 Esercizio 18

Listing 5: es18.m

```
%Codice esercizio 18
f = @(x)(cos((pi*(x.^2))/2));
x = linspace(-1, 1, 100001);
linerrors = zeros(1, 40);
chebyerrors = zeros(1, 40);
for n = 4:100
    xlin = linspace(-1, 1, n+1);
    xcheby = chebyshev(-1,1,n+1);
    %xcheby(1)=-1;
    %xcheby(n+1)=1;
    ylin = splinenat(xlin,f(xlin),x);
    ycheby = splinenat(xcheby,f(xcheby),x);
    ylin=ylin';
    ycheby=ycheby';
    linerrors(n) = norm(abs(f(x) - ylin), inf);
    chebyerrors(n) = norm( abs(f(x) - ycheby), inf);
end
semilogy(linerrors);
hold on;
semilogy(chebyerrors);
xlabel('numero di ascisse di interpolazione');
ylabel('massimo errore di interpolazione');
legend({'ascisse  equidistanti', 'ascisse di chebyshev'},'Location','northeast');
```

6.6 Esercizio 19

Listing 6: es19.m

```
%Codice esercizio 19
f = @(x)(cos((pi*(x.^2))/2));
x = linspace(-1, 1, 100001);
linerrors = zeros(1, 40);
chebyerrors = zeros(1, 40);
for n = 4:100
    xlin = linspace(-1, 1, n+1);
    xcheby = chebyshev(-1,1,n+1);
    ylin = spline(xlin,f(xlin),x);
    ycheby = spline(xcheby,f(xcheby),x);
    linerrors(n) = norm(abs(f(x) - ylin), inf);
    chebyerrors(n) = norm( abs(f(x) - ycheby), inf);
end
semilogy(linerrors);
hold on;
semilogy(chebyerrors);
xlabel('numero di ascisse di interpolazione');
ylabel('massimo errore di interpolazione');
legend({'ascisse  equidistanti', 'ascisse di chebyshev'},'Location','northeast');
```

6.7 Esercizio 20

Listing 7: es20.m

```
%Codice esercizio 20

```

```
f = @(x)(cos((pi*x.^2)/2));
fp = @(x)(f(x) + 10^(-3)*rand(size(x)));
xi = -1 + 2*(0:10^4)/10^4;
fi = f(xi);
fpi = fp(xi);
errors=zeros(1, 20);
for m = 1:20
    y = minimiquadrati(xi, fpi, m);
    errors(m) = norm(abs(y-fi), inf);
end
semilogy(errors);
xlabel('grado del polinomio');
ylabel('errore di interpolazione massimo');
```

6.8 Esercizio 21

Listing 8: es21.m

```
%Codice esercizio 21

for i = 1:7
    weights= rats(ncweights(i))
end
```

6.9 Esercizio 22

Listing 9: es22.m

```
%Codice esercizio 22

rapp = zeros(1, 50);
for i = 1:50
    rapp(i) = sum(abs(ncweights(i)))/i;
end
semilogy(rapp);
xlabel('grado n della formula di Newton-Cotes');
ylabel('^{K_n}/_{K}');
```

6.10 Esercizio 23

Listing 10: es23.m

```
%Codice esercizio 23

value = log(cos(1)/cos(1.1));
x = zeros(1,9);
errors=zeros(1, 9);
for i = 1:9
    x(i) = newtoncotes(@tan, -1,1.1, i);
    errors(i) = abs(value-x(i));
end
```

6.11 Esercizio 24

Listing 11: es24.m

```
%Codice esercizio 24

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

6.12 Esercizio 25

Listing 12: es25.m

```
%Codice esercizio 25

format long e
f = @(x)(1/(1+100*x.^2));
a = -1;
b = 1;
itrap = zeros(1, 5);
trap_points = zeros(1, 5);
isimp = zeros(1, 5);
simp_points = zeros(1, 5);
for i = 1:5
    [itrap(i), points] = adaptrap(f, a, b, 10^(-i-1));
    trap_points(i) =  length(points);
    [isimp(i), points] = adapsim(f, a, b, 10^(-i-1));
    simp_points(i) = length(points);
end
integrali = [itrap; isimp];
npoints = [trap_points; simp_points];
row_names = {'trapezi adattiva', 'simpson adattiva'};
colnames = {'10^-2','10^-3','10^-4','10^-5','10^-6'};
values = array2table(integrali,'RowNames',row_names,'VariableNames',colnames);
npoints = array2table(npoints,'RowNames',row_names,'VariableNames',colnames);
disp(values);
format
disp(npoints);
```