Elaborato di

Calcolo Numerico

Anno Accademico 2018/2019

Alessio Falai - 6134275 alessio.falai@stud.unifi.it
Leonardo Calbi - 6155786 leonardo.calbi@stud.unifi.it

May 23, 2019

Capitoli

1 Capitolo 1 **1**
1.1 Esercizio 1 1
1.2 Esercizio 2 2
1.3 Esercizio 3 3
1.4 Esercizio 4 4

2 Capitolo 2 **5**
2.1 Esercizio 5 5
2.2 Esercizio 6 8
2.3 Esercizio 7 9

3 Capitolo 3 **12**
3.1 Esercizio 8 12
3.2 Esercizio 9 12
3.3 Esercizio 11 13
3.4 Esercizio 12 14

4 Capitolo 4 **15**
4.1 Esercizio 14 15
4.2 Esercizio 15 16
4.3 Esercizio 16 17
4.4 Esercizio 19 18

5 Capitolo 5 **22**
5.1 Esercizio 22 22

6 Capitolo 6 **23**
6.1 Esercizio 24 23
6.2 Esercizio 26 23
6.3 Esercizio 27 25
6.4 Esercizio 28 26

1 Capitolo 1

1.1 Esercizio 1

Verificare che, per h sufficientemente piccolo:

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

Usando gli sviluppi di Taylor fino al secondo ordine otteniamo:

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

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

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

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

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

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

Effettuando le opportune sostituzioni, la relazione iniziale diventa:

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

Semplificando, si ottiene che:

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

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

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

Infine, otteniamo che:

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

Dunque, l'uguaglianza iniziale è verificata.

1.2 Esercizio 2

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

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

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

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

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

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

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

1.3 Esercizio 3

Eseguire il seguente *script* Matlab:

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

Spiegare i risultati ottenuti.

I risultati ottenuti:

```
1  u =
2          1.000000000000000e-300
3
4  u =
5          1.119916342203863e-300
```

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

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

Alcune premesse:

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

Alcune notazioni importanti:

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

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

1.4 Esercizio 4

Eseguire le seguenti istruzioni Matlab:

```
format long e
a=1.111111111111111
b=1.11111111111111
a+b
a-b
```

Spiegare i risultati ottenuti.

I risultati ottenuti:

```
a =
        1.111111111111111e+00

b =
        1.111111111111110e+00

ans =
        2.222222222222221e+00

ans =
        8.881784197001252e-16
```

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

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

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

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

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

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

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

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

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

2 Capitolo 2

2.1 Esercizio 5

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

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

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

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

essendo tol una opportuna tolleranza specificata in ingresso.

Metodo di bisezione:

```
function [x, i, imax] = bisection(f, a, b, tol)
%
% [x, i, imax] = bisection(f, a, b, tol) Determina uno zero della funzione in ingresso,
%                                        sull'intervallo [a, b],
%                                        utilizzando il metodo di bisezione.
%
fa = feval(f, a);
fb = feval(f, b);
if fa == 0
    x = a;
    return
end
if fb == 0
    x = b;
    return
end
if abs(fa) == Inf || abs(fb) == Inf
    error('Funzione non valutabile in uno degli estremi.')
    return
end
if fa * fb > 0
    error('Funzione non soddisfacente la condizione f(a) * f(b) < 0.')
    return
end

x = (a + b) / 2;
fx = feval(f, x);
imax = ceil(log2(b - a) - log2(tol * (1 + abs(x))));
for i = 2 : imax
    f1x = ((fb - fa) / (b - a));
    if abs(fx / f1x) <= tol * (1 + abs(x))
        break
    elseif fa * fx < 0
        b = x;
        fb = fx;
    else
```

```
        a = x;
        fa = fx;
    end
    x = (a + b) / 2;
    fx = feval(f, x);
    imax = ceil(log2(b - a) - log2(tol * (1 + abs(x))));
end
return
```

Metodo di Newton:

```
function [x, i] = newton(f, f1, x0, imax, tol)
%
% [x, i] = newton(f, f1, x0, imax, tol) Determina uno zero della funzione
%                                       in ingresso utilizzando il metodo di Newton.
%
fx = feval(f, x0);
f1x = feval(f1, x0);
x = x0 - fx / f1x;
i = 0;
go = 1;
while go && i < imax
    i = i + 1;
    x0 = x;
    fx = feval(f, x0);
    f1x = feval(f1, x0);
    if f1x == 0, error('La derivata prima ha assunto valore zero, impossibile continuare.
        '), end
    x = x0 - fx / f1x;
    go = abs(x - x0) > tol * (1 + abs(x));
end
if go, disp('Il metodo non converge.'), end
return
```

Metodo delle secanti:

```
function [x, i] = secant(f, f1, x0, imax, tol)
%
% [x, i] = secant(f, f1, x0, imax, tol) Determina uno zero della funzione
%                                       in ingresso utilizzando il
%                                       metodo delle secanti.
%
fx = feval(f, x0);
f1x = feval(f1, x0);
if f1x == 0, error('La derivata prima ha assunto valore zero, impossibile continuare.'),
    end
x = x0 - fx / f1x;
i = 0;
go = 1;
while go && i < imax
    i = i + 1;
    fx0 = fx;
    fx = feval(f, x);
    t = (fx - fx0);
    if t == 0, error('Impossibile determinare la radice nella tolleranza desiderata.'),
        end
    x1 = (fx * x0 - fx0 * x) / t;
    x0 = x;
    x = x1;
```

```
    go = abs(x - x0) > tol * (1 + abs(x));
end
if go, disp('Il metodo non converge.'), end
return
```

Metodo delle corde:

```
function [x, i] = chord(f, f1, x, imax, tol)
%
% [x, i] = chord(f, f1, x, imax, tol) Determina uno zero della funzione
%                                     in ingresso utilizzando il
%                                     metodo delle corde.
%
f1x = feval(f1, x);
if f1x == 0, error('La derivata prima ha valore nullo, impossibile continuare.'), end
go = 1;
i = 0;
while go && i < imax
    i = i + 1;
    x0 = x;
    fx = feval(f, x0);
    x = x0 - fx / f1x;
    go = abs(x - x0) > tol * (1 + abs(x));
end
if go, disp('Il metodo non converge.'), end
return
```

2.2 Esercizio 6

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

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

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

Codice Matlab

```
f = @(x) x - exp(-x) * cos(x / 100);
f1 = @(x) 1 + exp(-x) * cos(x / 100) + (exp(-x) * sin(x / 100)) / 100;
x0 = -1;
a = -1;
b = 1;
imax = 1000;
for i = 1 : 12
        tol = 10^(-i);
        rtol=['Tolleranza: ',num2str(tol, '%e')];
        disp(rtol)
        [xB, it] = bisection(f, a, b, tol);
        disp(['Bisezione: ', num2str(xB, '%10.12e'), ' Iterazioni ', num2str(it)])
        [xN, it] = newton(f, f1, x0, imax, tol);
        disp(['Newton: ', num2str(xN, '%10.12e'), ' Iterazioni ', num2str(it)])
        [xC, it] = chord(f, f1, x0, imax, tol);
        disp(['Corde: ', num2str(xC, '%10.12e'), ' Iterazioni ', num2str(it)])
        [xS, it] = secant(f, f1, x0, imax, tol);
        disp(['Secanti: ', num2str(xS, '%10.12e'), ' Iterazioni ', num2str(it)])
        disp(' ')
end
```

Risultati

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

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

2.3 Esercizio 7

Calcolare la molteplicità della radice nulla della funzione

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

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

Metodo di Newton modificato:

```
function [x, i] = modnewton(f, f1, x0, m, imax, tol)
%
% [x, i] = modnewton(f, f1, x0, m, imax, tol) Determina uno zero della funzione
%                                             in ingresso, con molteplicita' multipla,
%                                             utilizzando il metodo di Newton modificato.
%
fx = feval(f, x0);
f1x = feval(f1, x0);
x = x0 - m * (fx / f1x);
i = 0;
go = 1;
while go && i < imax
    i = i + 1;
    x0 = x;
    fx = feval(f, x0);
    f1x = feval(f1, x0);
    if f1x == 0, error('La derivata prima ha assunto valore zero, impossibile continuare.
        '), end
    x = x0 - m * (fx / f1x);
    go = abs(x - x0) > tol * (1 + abs(x));
end
if go, disp('Il metodo non converge.'), end
return
```

Metodo di accelerazione di Aitken:

```
function [x, i] = aitken(f, f1, x0, imax, tol)
%
% [x, i] = aitken(f, f1, x0, imax, tol) Determina uno zero della funzione
%                                       in ingresso, con molteplicita' multipla,
%                                       utilizzando il metodo di Aitken.
%
x = x0;
i = 0;
go = 1;
while go && i < imax
    i = i + 1;
    x0 = x;
    fx = feval(f, x0);
    f1x = feval(f1, x0);
    if f1x == 0, error('La derivata prima ha assunto valore zero, impossibile continuare.
        '),end
    x1 = x0 - fx / f1x;
    fx = feval(f, x1);
    f1x = feval(f1, x1);
```

```
    if f1x == 0
        if abs(x1 - x0) < tol * (1 + abs(x))
            x = x1;
            disp('Nella tolleranza richiesta, ma con approssimazioni intermedie al metodo
                di Aitken.');
            return
        end
        error('La derivata prima ha assunto valore zero, impossibile continuare.')
    end
    x = x1 - fx / f1x;
    t = (x - 2 * x1 + x0);
    if t == 0
        if abs(x - x1) < tol * (1 + abs(x))
            disp('Nella tolleranza richiesta, ma con approssimazioni intermedie al metodo
                di Aitken.');
            return
        end
        error('Impossibile determinare la radice nella tolleranza desiderata.')
    end
    x = (x * x0 - (x1)^2) / t;
    go = abs(x - x0) > tol * (1 + abs(x));
end
if go, disp('Il metodo non converge.'), end
return
```

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

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

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

Codice Matlab

```
f = @(x) x^2 * sin(x^2);
f1 = @(x) 2 * x * (sin(x^2) + x^2 * cos(x^2));
x0 = 1;
imax = 1000;
m = 3;
for i = 1 : 12
        tol = 10^(-i);
        rtol=['Tolleranza: ', num2str(tol, '%e')];
        disp(rtol)
        [xN, it] = newton(f, f1, x0, imax, tol);
        disp(['Newton: ', num2str(xN, '%10.12e'), ' Iterazioni ', num2str(it)])
        [xNM, it] = modnewton(f, f1, x0, m, imax, tol);
        disp(['Newton Modificato: ', num2str(xNM, '%10.12e'), ' Iterazioni ', num2str(it)
            ])
        [xA, it] = aitken(f, f1, x0, imax, tol);
        disp(['Aitken: ', num2str(xA, '%10.12e'), ' Iterazioni ', num2str(it)])
        disp(' ')
end
```

Risultati

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

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

3 Capitolo 3

3.1 Esercizio 8

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

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

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

Fattorizzazione LU con pivoting parziale:

```
function [A, p] = palu (A)
%
% function [A, p] = palu (A) calcola la fattorizzazione LU con pivoting parziale e
%                            restituisce la matrice fattorizzata e il vettore
%                            contentente l'informazione relativa alla matrice
%                            di permutazione
%
[m, n] = size(A);
if m ~= n
    error('Matrice non quadrata.');
end
p = 1 : n;
for i = 1 : n - 1
    [mi, ki] = max(abs(A(i : n, i)));
    if mi == 0
        error('Matrice singolare.')
    end
    ki = ki + i - 1;
    if ki > i
        A([i ki], :) = A([ki i], :);
        p([i ki]) = p([ki i]);
    end
    A(i + 1 : n, i) = A(i + 1 : n, i) / A(i, i);
    A(i + 1 : n, i + 1 : n) = A(i + 1 : n, i + 1 : n) - A(i + 1 : n, i) * A(i, i + 1 : n)
        ;
end
p = p';
return
```

3.2 Esercizio 9

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

$$functionx = lusolve(LU, p, b)$$

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

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

```
function x = palusolve(LU, p, b)
%
% x = palusolve(LU, p, b) risolve il sistema lineare LUx = b,
%                         con LU matrice fattorizzata LU
%                         con pivoting parziale
%
x = b;
[m, n] = size(LU);
if n ~= length(x) || m ~= n, error(Dati inconsistenti.), end
x = infsolve(LU, x(p));
x = supsolve(LU, x);
return

function [x] = infsolve(L, x)
n = length(x);
for i = 1:n
    x(i + 1 : n) = x(i + 1 : n) - L(i + 1 : n, i) * x(i);
end
return

function [x] = supsolve(U, x)
n = length(x);
for i = n : -1 : 1
    x(i) = x(i) / U(i, i);
    x(1 : i - 1) = x(1 : i - 1) - U(1 : i - 1, i) * x(i);
end
return
```

3.3 Esercizio 11

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

$$function\ QR = myqr(A)$$

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

Fattorizzazione QR:

```
function QR = myqr(A)
%
% QR = myqr(A) calcola la fattorizzazione QR della
%              matrice sovradeterminata A
%
[m, n] = size(A);
if m < n, error('Numero di righe della matrice minore del numero di colonne.'); end
QR = A;
for i = 1 : n
    alfa = norm(QR(i : m, i));
    if alfa == 0, error('La matrice A 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
return
```

3.4 Esercizio 12

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

$$function\ x = qrsolve(QR, b)$$

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

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

```
function x = qrsolve(QR, b)
%
% x = qrsolve(QR, b) Risolve sistemi lineari data la matrice QR
%                    contenente le parti significative di una fattorizzazione
%                    QR di Householder, che computa g = Q.'b e in seguito Rx = g
%
B = QR;
[m, n] = size(B);
if m < n || m ~= length(b), error('Sistema non risolvibile.'); end
x = b(:);
for i = 1 : n
    v = [1; B(i + 1 : m, i)];
    beta = 2 / (v' * v);
    x(i : m) = x(i : m) - (beta * (v' * x(i : m))) * v;
end
r = norm(x(n + 1 : m));
x = x(1 : n);
for i = n : -1 : 1
    x(i) = x(i) / B(i, i);
    x(1 : i - 1) = x(1 : i - 1) - B(1 : i - 1, i) * x(i);
end
return
```

4 Capitolo 4

4.1 Esercizio 14

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

Polinomio interpolante di Lagrange:

```
function y = lagrange(xi, fi, x)
%
% y = lagrange(xi, fi, x) calcola il polinomio interpolante le coppie (xi, fi)
%                         sulle ascisse x
%
n = length(xi); if length(fi) ~= n, error(Dati inconsistenti.), end
for i = 1 : n - 1
    for j = i + 1 : n
        if xi(i) == xi(j), error(Ascisse non distinte.), end
    end
end
y = zeros(size(x));
for i = 1 : n
    if fi(i) ~= 0
        li = 1;
        for k = [1 : i - 1 i + 1 : n]
            li = li .* (x - xi(k)) / (xi(i) - xi(k));
        end
        y = y + fi(i) * li;
    end
end
return
```

Polinomio interpolante di Newton:

```
function y = newton(xi, fi, x)
%
% y = newton(xi, fi, x) calcola il polinomio interpolante le coppie (xi, fi)
%                       sulle ascisse x
%
n = length(xi); if length(fi) ~= n, error(Dati inconsistenti.), end
for i = 1 : n - 1
    for j = i + 1 : n
        if xi(i) == xi(j), error(Ascisse non distinte.), end
    end
end
fi = divdif(xi, fi);
y = fi(n) * ones(size(x));
for i = n - 1 : -1 : 1
    y = y .* (x - xi(i)) + fi(i);
end
return

function fi = divdif(xi, fi)
%
% f = divdif(xi, fi) calcola le differenze divise
%                    sulle coppie (xi, fi)
```

```
%
n = length(xi);
for i = 1 : n - 1
    for j = n : -1 : i + 1
        fi(j) = (fi(j) - fi(j - 1)) / (xi(j) - xi(j - i));
    end
end
return
```

4.2 Esercizio 15

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

Polinomio interpolante di Hermite:

```
function y = hermite(xi, fi, f1i, x)
%
% y = hermite(xi, fi, f1i, x) Calcola il polinomio interpolante di grado n
%                             in forma di Hermite, nei punti x
%
n = length(xi) - 1;
xh = zeros(2 * n + 2, 1);
xh(1 : 2 : 2 * n + 1) = xi;
xh(2 : 2 : 2 * n + 2) = xi;
fh = zeros(2 * n + 2, 1);
fh(1 : 2 : 2 * n + 1) = fi;
fh(2 : 2 : 2 * n + 2) = f1i;
fh = divdif(xh, fh);
y = horner(xh, fh, x);
y = y.';
return

function fh = divdif(xh, fh)
%
% fh = divdif(xh, fh) Calcola le differenze divise
%                     sulle coppie (xh, fh)
%
nh = length(xh) - 1;
for i = nh : -2 : 3
    fh(i) = (fh(i) - fh(i - 2)) / (xh(i) - xh(i - 1));
end
for i = 2 : nh
    for j = nh + 1 : -1 : i + 1
        fh(j) = (fh(j) - fh(j - 1)) / (xh(j) - xh(j - i));
    end
end
return

function y = horner(xh, fh, x)
%
% y = horner(xh, fh, x) Valuta il polinomio interpolante nelle ascisse xh,
%                       se il vettore fh contiene i coefficienti di p,
%                       ordinati a partire dal termine noto
%
```

```
nh = length(xh) - 1;
y = fh(nh + 1) * ones(size(x));
for i = nh : -1 : 1
    y = y .* (x - xh(i)) + fh(i);
end
return
```

4.3 Esercizio 16

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

Spline cubica naturale:

```
function y = spline3(xi, fi, x)
%
% y = spline3(xi, fi, x) Calcola la spline cubica naturale
%                        su una partizione assegnata e
%                        la valuta nelle ascisse date
%
if any(size(xi) ~= size(fi)) || size(xi,2) ~= 1
    error('xi e fi devono essere vettori della stessa lunghezza.');
end

n = length(xi);

h = xi(2 : n) - xi(1 : n - 1);
d = (fi(2 : n) - fi(1 : n - 1)) ./ h;

lower = h(2 : end);
main  = 2 * (h(1 : end - 1) + h(2 : end));
upper = h(1 : end - 1);

T = spdiags([lower main upper], [-1 0 1], n - 2, n - 2);
rhs = 6 * (d(2 : end) - d(1 : end - 1));

m = T\rhs;

m = [ 0; m; 0];

s0 = fi;
s1 = d - h .* (2 * m(1 : end - 1) + m(2 : end)) / 6;
s2 = m / 2;
s3 =(m(2 : end) - m(1 : end - 1)) ./ (6 * h);

l = length(x);
y =zeros(l, 1);
for j = 1 : l
    if x(j) < xi(1) || x(j) > xi(n)
        error('Funzione interpolante non valutabile nelle ascisse date.')
    end
    i = 1;
    while xi(i) < x(j), i = i + 1; end
    i = i - 1;
```

```
    y(j) = s0(i) + s1(i) * (x(j) - xi(i)) + s2(i) * (x(j) - xi(i)) .^2 + s3(i) * (x(j) -
        xi(i)) .^3;
end
return
```

4.4 Esercizio 19

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

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

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

Costante di Lebesgue:

```
function leb = lebesgue(x)
%
% leb = lebesgue(x) Calcola il valore della costante di Lebesgue,
%                   dato il vettore x di N nodi (ordinato)
%
n = length(x);
x_g = linspace(x(1), x(end), 10001);
m = length(x_g);
vLeb = zeros(m, 1);
for i = 1 : m
    for j = 1 : n
        range = [1 : j - 1, j + 1 : n];
        bl = prod(x_g(i) - x(range)) / prod(x(j) - x(range));
        vLeb(i) = vLeb(i) + abs(bl);
    end
end
leb = norm(vLeb, inf);
end
```

Ascisse di Chebyshev:

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

Codice Matlab

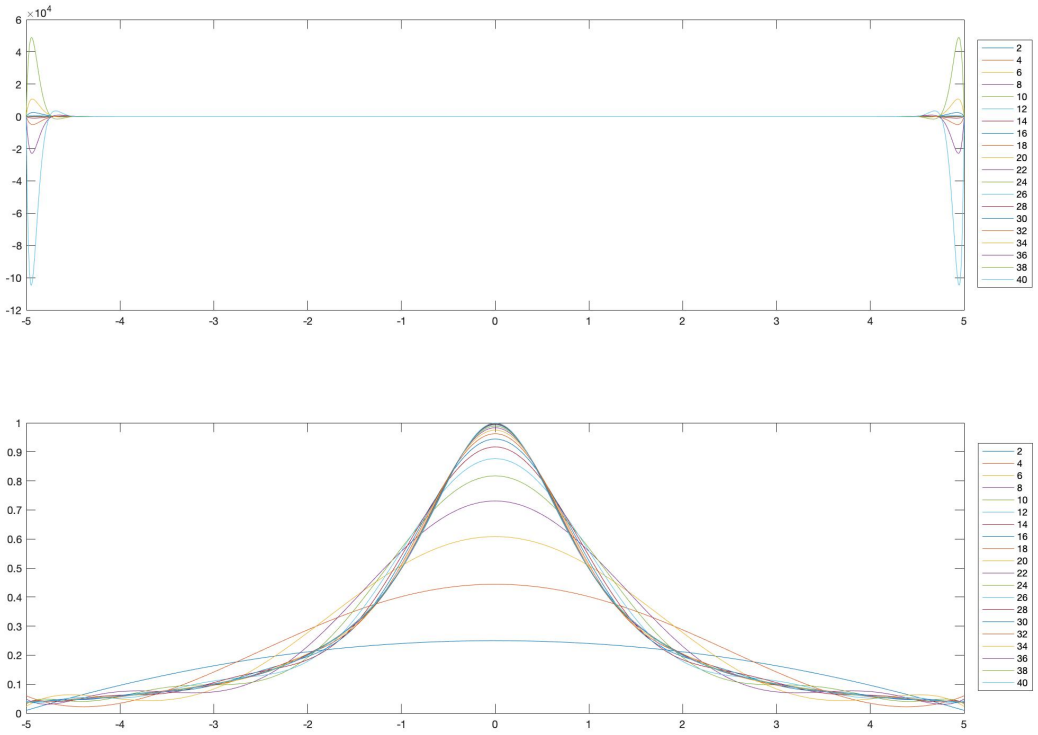
```
f = @(x) 1 ./ (1 + x.^2);
a = -5;
```

```
b = 5;
n = 2:2:40;
kxi = zeros(length(n), 1);
kci = zeros(length(n), 1);
x = linspace(a, b, 10001);

ax1 = subplot(2,1,1);
for i = 1 : length(n)
        xi = linspace(a, b, n(i) + 1);
        fxi = f(xi);
        yxi = lagrange(xi, fxi, x);
        plot(ax1, x, yxi)
        hold on
        kxi(i, 1) = lebesgue(xi);
end
legend('2','4','6','8','10','12','14','16','18','20','22','24','26','28','30','32','34','
    36','38','40')
hold off

ax2 = subplot(2,1,2);
for i = 1 : length(n)
        ci = ceby(n(i) + 1, a, b);
        fci = f(ci);
        yci = lagrange(ci, fci, x);
        plot(ax2, x, yci)
        hold on
        kci(i, 1) = lebesgue(ci);
end
legend('2','4','6','8','10','12','14','16','18','20','22','24','26','28','30','32','34','
    36','38','40')
hold off
```

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



Polinomio di Lagrange con n° di ascisse $[2, 4, 6, 8, .., 40]$

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

Costante di Lebesgue con ascisse equispaziate

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

Costante di Lebesgue con ascisse di Chebyshev

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

5 Capitolo 5

5.1 Esercizio 22

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

Formule adattattive dei trapezi:

```
function I4 = adaptrap(f, a, b, tol, fa, fb)
%
% I4 = adaptrap(f, a, b, tol, fa, fb) calcola l'integrale definito di f
%                                     in [a, b], utilizzando le formule
%                                     adattative dei trapezi
%
if nargin <= 4
    fa = feval(f, a);
    fb = feval(f, b);
end
h = b - a;
x1 = (a + b) / 2;
f1 = feval(f, x1);
I1 = (h / 2) * (fa + fb);
I2 = (I1 + h * f1) / 2;
e = abs(I2 - I1) / 3;
if e > tol
    I2 = adaptrap(f, a, x1, tol / 2, fa, f1) + adaptrap(f, x1, b, tol / 2, f1, b);
end
return
```

Formule adattattive di Simpson:

```
function I4 = adapsimp(f, a, b, tol, fa, fb)
%
% I4 = adapsimp(f, a, b, tol, fa, fb) calcola l'integrale definito di f
%                                     in [a, b], utilizzando le formule
%                                     adattative di Simpson
%
if nargin <= 4
    fa = feval(f, a);
    fb = feval(f, b);
end
h = b - a;
x2 = (a + b) / 2;
f2 = feval(f, x2);
I2 = (h / 6) * (fa + 4 * f2 + fb);
x1 = (a + x2) / 2;
f1 = feval(f, x1);
x3 = (x2 + b) / 2;
f3 = feval(f, x3);
I4 = (h / 12) * (fa + 4 * f1 + 2 * f2 + 4 * f3 + fb);
e = abs(I4 - I2) / 15;
if e > tol
    I4 = adapsimp(f, a, x2, tol / 2, fa, f2) + adapsimp(f, x2, b, tol / 2, f2, fb);
end
return
```

6 Capitolo 6

6.1 Esercizio 24

Scrivere una function che implementi efficientemente il metodo delle potenze.

Metodo delle potenze:

```
function [lambda, i] = power(A, tol, x0, imax)
%
% [lambda, i] = power(A, tol, x0, imax) calcola l'autovalore dominante
%                                       semplice della matrice in input,
%                                       utilizzando il metodo delle potenze
%
n = size(A, 1);
if nargin <= 2
    x = rand(n,1);
else
    x = x0;
end
x = x / norm(x);
if nargin <= 3
    imax = 100 * n * round( - log(tol));
end
lambda = inf;
for i = 1 : imax
    lambda0 = lambda;
    v = A * x;
    lambda = x' * v;
    err = abs(lambda - lambda0);
    if err <= tol
        break;
    end
    x = v / norm(v);
end
if err > tol
    error('Autovalore non trovato nella tolleranza specificata.');
end
return
```

6.2 Esercizio 26

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

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

```
function [x, i] = itersolve(M, N, b, x0, tol, imax)
%
```

```
% [x, i] = itersolve(M, N, b [,x0 [, tol [, imax]]]) risolve il sistema lineare, con
%                                                     matrice dei coefficienti definita
%                                                     da un generico splitting M - N, in
%                                                     modo iterativo
%
n = length(b);
if rank(M) ~= n, error('Matrice M singolare.'), end
if nargin <= 5
    if nargin <= 4
        if nargin <= 3
            x0 = rand(n, 1);
        end
        tol = 1E-6;
    elseif tol <= 0 || tol >= 0.1
        error('Tolleranza inconsistente.');
    end
    imax = n * max(1, ceil(-log10(tol))) * 100;
end
x = x0;
A = M - N;
for i = 1 : imax
    r = A * x - b;
    nr = norm(r, inf);
    if nr <= tol, break, end
    x = x - M\r;
end
if nr > tol, warning('Soluzione non trovata nella tolleranza specificata.'), end
return
```

Risoluzione iterativa del sistema lineare con il metodo msolve:

```
function [x, i] = splitting(b, matvec,msolve, x0, tol, imax)
%
% [x, i] = itersolve(b, matvec, msolve [,x0 [, tol [, imax]]]) risolve il sistema
%                                                               lineare con il metodo
%                                                               iterativo definito dalla
%                                                               function msolve
%
n = length(b);
if nargin <= 5
    if nargin <= 4
        if nargin <= 3
            x0 = zeros(n, 1);
        end
        tol = 1E-6;
    elseif tol <= 0 || tol >= 0.1
        error('Tolleranza inconsistente.');
    end
    imax = n * max(1, ceil(-log10(tol))) * 100;
end
x = x0;
tolb = tol * norm(b);
for i = 1 : imax
    r = matvec(x) - b;
    nr = norm(r);
    if nr <= tolb, break, end
    x = x - msolve(r);
end
if nr > tolb, warning('Soluzione non trovata nella tolleranza specificata.'), end
```

```
29 return
```

6.3 Esercizio 27

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

Metodo di Jacobi:

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

Metodo di Gauss-Seidel:

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

6.4 Esercizio 28

Con riferimento alla matrice A_{N} definita in (1), risolvere il sistema lineare

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

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

Prodotto *ad hoc* matrice vettore:

```
function y = matvec(x)
%
% y = matvec(x) calcola il prodotto ad hoc
%               matrice vettore
%
y = 4 * x;
y(1 : end - 1) = y(1 : end - 1) - x(2 : end);
y(2 : end) = y(2 : end) - x(1 : end - 1);
y(1 : end - 8) = y(1 : end - 8) - x(9 : end);
y(9 : end) = y(9 : end) - x(1 : end - 8);
return
```

Metodo di Jacobi:

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

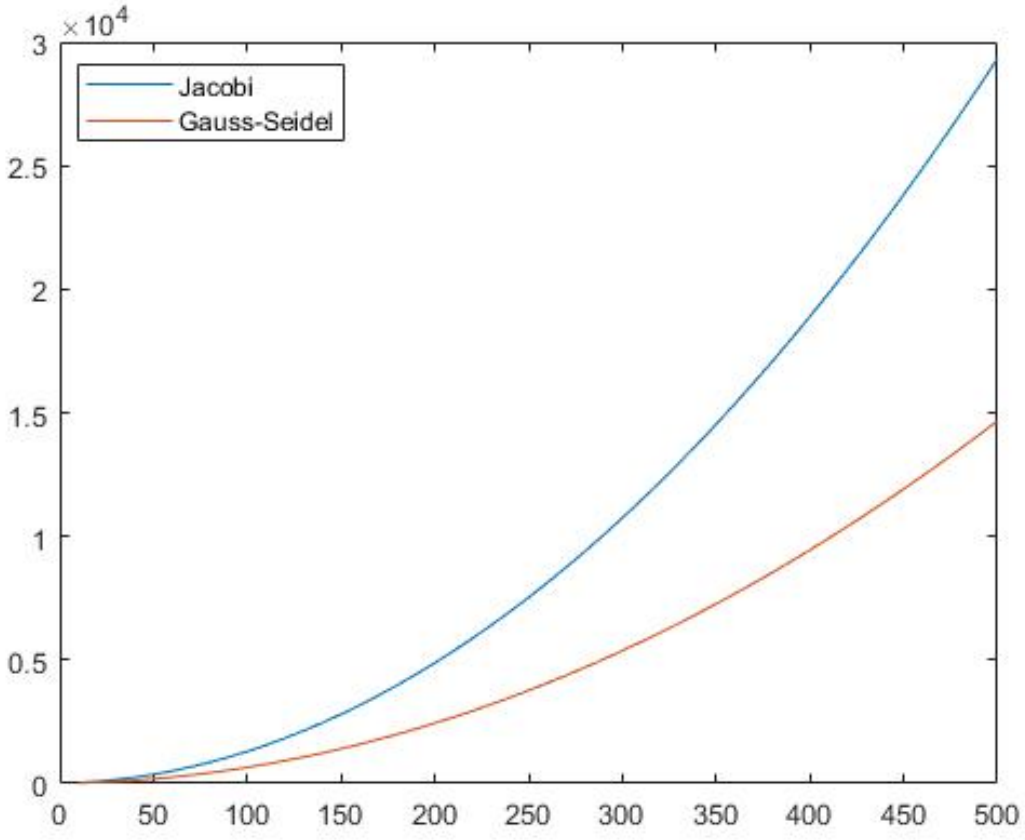
Metodo di Gauss-Seidel:

```
function u = gs(r)
%
% u = gs(r) risolve il sistema lineare ad hoc
%           con il metodo iterativo di Gauss–Seidel
%
n = length(r);
if n < 10, error('Dimensione minima non rispettata.'), end
u = r;
u(1) = u(1) / 4;
for i = 2 : 8
    u(i) = (u(i) + u(i - 1)) / 4;
end
for i = 1: n - 8
    u(i + 8) = (u(i + 8) + u(i) + u(i + 7)) / 4;
end
return
```

Codice Matlab

```
x = linspace(10, 500, 50);
tol = 1E − 8;
ij = zeros(50, 1);
igs = zeros(50, 1);
for n = 10 : 10 : 500
    b = ones(n, 1);
    x0 = zeros(n, 1);
    [x, i] = splitting(b, @matvec, @jacobi, x0, tol);
    ij(n / 10) = i;
    [x, i] = splitting(b, @matvec, @gs, x0, tol);
    igs(n / 10) = i;
end
plot(x, ij, x, igs);
legend('Jacobi', 'Gauss–Seidel');
```

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



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