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 9, 2019

Capitoli

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:

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

```
u =
        1.000000000000000e−300

u =
        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
```

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 **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
```