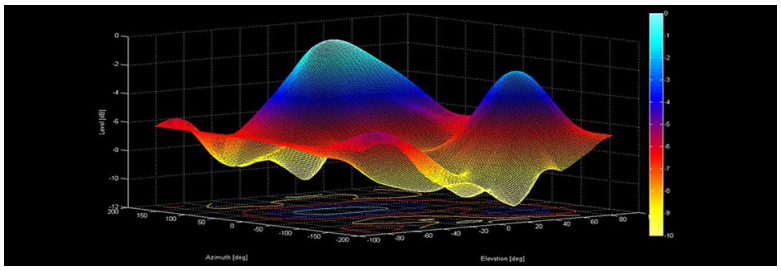
Relazione per il corso di Calcolo Numerico

Prof. Luigi Brugnano

Prof. Cesare Bracco



Bajron Ismailaj
Matricola 2686563
Esame da 6 crediti
Secondo Anno
Vecchio ordinamento
01/07/2020

Indice

Sommario

Capitolo 1

Esercizi capitolo 1

1.1 Esercizio 1

Verificare che, per h sufficientemente piccolo,

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

Soluzione

La dimostrazione si basa su gli sviluppi di Taylor con il resto di Lagrange.
Sia f una funzione derivabile $n+1$ volte in $[a,b]$. con derivata $f^{(n+1)}$ continua e preso h tale che f sia definito nel intervallo $[x_0-h; x_0+h]$ allora vale la seguente,

$$f(x+h) = \sum_{k=0}^{n} \frac{f^{(k)}x_0}{k!}(x-x_0)^k + R_n(x)$$

dove $R_n(x)$ è il resto in forma di Lagrange per ogni $x \in [a,b]$ esiste un numero ξ compreso fra x_0 ed x, tale che:

$$R_n(x) = \frac{f^{(n)}(\xi)}{(n+1)!}(x-x_0)^{n+1}$$

Si sviluppa i polinomi di Taylor di terzo ordine per la funzione centrata in x_0,

$$f(x) = f(x_0) + f'(x_0)(x-x_0) + \frac{1}{2}f''(x_0)(x-x_0)^2 + \frac{1}{6}f'''(x_0)(x-x_0)^3 + O((x-x_0)^4)$$

dal quale segue

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

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

sostituendo i risultati ottenuti nelle equazione principale e semplificando si ottiene la seguente,

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

1.2 Esercizio 2

Eseguire il seguente script Matlab, e spiegare cosa calcola.

```
u = 1; while 1, if 1+u==1, break, end, u = u/2; end, u
```

Soluzione

Partendo da un valore $u = 1$ il codice utilizza un ciclo *while* dove ad ogni passo divide u per 2. Il ciclo termina se e vera la condizione $1 + u = 1$. La condizione d'uscita del ciclo ci dice sostanzialmente che si esce quando u diventa così piccolo che la sua somma con 1 viene avvertita dal calcolatore come qualcosa uguale di 1, cioè per $u < eps$ dove si è in presenza di underflow. Sapendo che *eps* la precisione di macchina rappresenta il più piccolo numero della mantissa:

$$eps = \frac{1}{2} b^{1-m}$$

si ha che u assume il valore $eps/2$ e per questo che la sua somma con 1 non viene considerato come un numero diverso da 1
Non a caso si esce dal ciclo dopo 53 iterazioni, la grandezza della mantissa dello standard IEEE 753 in doppia precisione con 52 bit per la frazione.

1.3 Esercizio 3

Eseguire il seguente script Matlab:

```
a = 1e20; b = 100; a-a+b
a = 1e20; b = 100; a+b-a
```

Spiegare i risultati ottenuti.

Soluzione

Nella prima linea di codice si ottiene come risultato il valore 100 invece eseguendo la seconda linea del codice si ottiene 0. Siamo in presenza di un classico esempio in qui in aritmetica finita la proprietà associativa della somma non vale. Questo perché quando il calcolatore esegue le operazioni $a - a + b$ per prima cosa esegue la differenza $a - a$ che vale 0 per poi sommare b ottenendo il risultato b aspettato.

Nel caso delle operazioni della seconda linea del codice dove $a + b - a$ il calcolatore cerca per prima di effettuare la somma fra $a + b$ dove si ha la somma fra un numero molto grande a e b un numero piccolo in confronto ad a. Visto che a è molto grande la sua rappresentazione in numero di macchina avviene con dell'errore di conversione poiché la mantissa perde precisione per il fatto che si ha un esponente molto grande infatti in doppia precisione il massimo intero rappresentato esattamente è $c = 2^{53}, a >> c$. Invece b si rappresenta esattamente in numero di macchina. Quando avviene la somma con b quest'ultima non influenza sul valore di a perché non abbastanza grande da influire su le cifre di precisione della mantissa di a. Il calcolatore considera la somma $a + b = a$ e quando sottrae a si ottiene 0. Si ha perdita di informazione per perdita di precisione.

1.4 Esercizio 4

Scrivere una function Matlab, $radn(x, n)$ che, avendo in ingresso un numero positivo x ed un intero n, ne calcoli la radice n-esima con la massima precisione possibile.

Soluzione

Per trovare la soluzione y tale che $y = \sqrt[n]{x}$ significa trovare un y tale che $y^n = x$. Trovare una soluzione del problema significa trovare gli zeri della funzione $f(y) = y^n - x$. La funzione $f(x)$ è ben definita e continua nel intervallo ed e derivabile per x positivo e n intero.
Visto la facilita del calcolo della derivata prima di $f(y)$ che vale $f'(y) = ny^{n-1}$ il metodo di interazione Newton è ideale per trovare lo zero della funzione. Una approssimazione con il metodo di Newton per una funzione $f(y)$ si basa su la seguente iterazione

$$y_{i+1} = y_i - \frac{f(y_i)}{f'(y_i)}$$

sostituendo si ha ,

$$y_{i+1} = y_i - \frac{y_i^n - x}{ny^{n-1}} = \frac{ny_i^n - y_i^n + x}{ny_i^{n-1}} = \frac{1}{n}\left(\frac{(n-1)y_i^n + x}{y_i^{n-1}}\right) = \frac{1}{n}\left(\frac{(n-1)y_i^n}{y_i^{n-1}} + \frac{x}{y_i^{n-1}}\right) = \frac{1}{n}\left((n-1)y_i + \frac{x}{y_i^{n-1}}\right)$$

La procedura ha successo se il denominatore è diverso da zero $x \neq 0$ inoltre anche $n \neq 0$.
Trovato il metodo di iterazione si determina il criterio di arresto. Per una massima precisione si considera la tolleranza uguale alla precisione macchina $toly = eps$ e si usa il criterio di arresto su la tolleranza relativa sull'accuratezza dell'approssimazione. In tal caso, il criterio di arresto diviene

$$\frac{|y_{i+1} - y_1|}{1 + |y_{i+1}|} \leq eps$$

In seguito la *function* **radn** in Matlab

```
function [ y ] = radn(x,n)
% [ y ] = radn(x,n)
%Il metodo rappresenta una procedura iterativa basatu sul metodo di Newton.
%Trova la radice n-essima di x. Dove x reale positivo e n interno.
% Input:
% - x: radicando della radice (deve essere maggiore di 0)
% - n: radice interno diverso da 0.
% Output:
% - y: approssimazione della radice
        if x <= 0 % se x reale non positivo esci con errore
                error('x deve essere > 0');
        end
        if n==0 %La divisione per 0 fa si che il metodo non funziona
                error('n deve essere interno diverso da 0');
        end
        yi=1; %Punto di inesco del metodo
        i = 1;  %Numero di iterazioni del metodo.
        while 1 %Ciclo al infinito finche non si arriva alla tolleranza giusta
                y=yi + (yi.^(1-n).*(x - yi.^n))./n;
                fprintf('\nx%d = %.16f' , i, y);
                errOnX = abs(y - yi)/(1 + abs(y)); %Tolleranza relativa
                if (errOnX <= eps) %eps precisione macchina
                        fprintf('\n\nIl metodo converge a %.16f\nNumero di iterazioni: %d\n', y, i
                            );
                        return;
                end
                yi = y;
                i = i + 1;
        end
end
```

La **function** utilizza come punto d'innesco $x = 1$ e ha un contatore del numero delle iterazioni eseguite memorizzate in i. Il ciclo interno che può essere migliorato aggiungendo un numero massimo di iterazioni in tal modo da prevenire cicli infiniti ma visto che si tratta di una funzione regolare e derivabile per una massima precisione si fa uso della condizione corrente.

1.5 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

$$|x_{i+1} - x_i| \leq tol * (1 + |x_1|)$$

essendo *tol* una opportuna tolleranza specificata in ingresso.

Soluzione

Per il criterio di arresto rivisitato si utilizza la tolleranza relativa per l'accuratezza dell'approssimazione.

$$\frac{|x_{i+1} - x_1|}{1 + |x_{i+1}|} \leq tolx$$

I metodi prendono come input il punto di innesco $x_0 = 0$, *iMax* numero massimo di iterazioni, *tol* tolleranza del errore e la funzione $f(x)$.

Metodo di Bisezione come argomento prende anche gli estremi del intervallo ma non usa un punto d'innesco.

```
function[ x ] = bisezione( a, b, f, iMax, tol)
% [ x ] = bisezione( a, b, f, iMax, tol)
% Il metodo di Bisezione con criterio d'arresto sul errore relativo. Per
% funzionare il metodo ha come requisito il fatto che il punto di zero x0
% della funzione deve cadere del intervallo, a<x0<b.
% - a: Punto  di estremo sinstro del intervallo [a,b]
% - b: Punto  di estremo desctro del intervallo [a,b]
% - f: stringa con il nome della funzione che implementa la funzione
% - iMax: numero massimo di iterazioni
% - tol: tolleranza
% Output:
% - x: approssimazione della radice
fa = feval(f,a);
fb = feval(f,b);
nVal = 2;
x0=b; % Inizializzazione xo
i = 1;
while (i < iMax )
        x = (a + b) / 2;
        fprintf('\nx%d = %1.16f', i, x);
        fx = feval(f, x);
        nVal = nVal + 1;
        errorX = abs(x - x0)/(1 + abs(x));
        if (errorX <= tol)
                fprintf('\nIl metodo converge a %.16f\nNumero di iterazioni: %d Numero di
                    valutazioni: %d\n', x, i,nVal);
                return;
        elseif fa * fx < 0
                b = x;
                fb = fx;
                else
                a = x;
                fa = fx;
        end
        x0=x;
i=i+1;
end
fprintf('\nIl metodo non converge! Numero di iterazioni: %d\nNumero di valutazioni: %d\n\n', i,
    nVal);
end
```

Per un giusto funzionamento la *function* richiede che lo zero $f(x) = 0$ appartiene al intervallo dato, $x \in [a, b]$. Vengono stampate sul console i risultati intermedi per la ricerca del zero. Alla fine stampa il numero delle iterazioni e delle valutazioni richieste.

Metodo di Newton che in più ha come argomento la deritava prima di $f(x)$.

```
function [ x ] = newton(x0,f,df, iMax, tol)
% [ x ] = newton(x0,f,df, iMax, tol)
% Metodo Newton con criterio d'arresto sul errore relativo
% Input:
% - x0: punto d'innesco
% - f: stringa con il nome della funzione che implementa la funzione
% - df: stringa con il nome della funzione che implementa la derivata della funzione
% - iMax: numero massimo di iterazioni
% - tol: tolleranza richiesta
% Output:
% - x: approssimazione della radice
fx = feval(f, x0);
dfx = feval(df, x0);
x = x0 - (fx / dfx);
i = 1;
nVal = 2;
while (i < iMax )
        fx = feval(f, x0);
        dfx = feval(df, x0);
        x = x0 - (fx / dfx);
        fprintf('\nx%d = %.16f ' , i, x);
        nVal = nVal + 2;
        errorX = abs(x - x0)/(1 + abs(x));
        if (errorX <= tol)
                fprintf('\nIl metodo converge a %.16f\nNumero di iterazioni: %d\nNumero
                    Valutazioni: %d \n', x, i, nVal);
                return;
        end
        x0 = x;
        i = i + 1;
end
fprintf('\nIl metodo non converge!\nNumero di iterazioni: %d\n %.16f', i , x );
end
```

La *function* prende come input anche la derivata della funzione ed ad ogni iterazione stampa il valore della radice della funzione $f(x)$ e stampa il numero delle iterazioni e delle valutazioni richieste.

Metodo delle secanti

```
function [ x ] = secanti(x0,fx,df, iMax, tol)
%  [ x ] = secanti(x0,fx,df, iMax, tol)
% Il metodo  delle Secanti con criterio d'arresto sul errore relativo
% Input:
% - x0: punto di nnesco
% - f: stringa con il nome della funzione che implementa la funzione
% - df: stringa con il nome della funzione che implementa la derivata della funzione
% - iMax: numero massimo di iterazioni
% - tol: tolleranza
% Output:
% - x: approssimazione della radice
fx0= feval(fx,x0);
dfx0 = feval(df, x0);
nVal=2;
x1 = x0 - (fx0 / dfx0);
i = 1;
while i < iMax
        fx1= feval(fx,x1);
        x =( fx1 * x0 - fx0 * x1 )/(fx1 - fx0);
        nVal =nVal + 1 ;
        fprintf('\nx%d = %.16f ' , i, x);
        errOnX = abs(x - x1)/(1 + abs(x));
        if (errOnX <= tol)
                fprintf('\nIl metodo converge a %.16f\nNumero di iterazioni: %d\nNumero
                    Valutazioni: %d \n', x, i, nVal);
                return;
        end
```

```
        x0 = x1;
        x1=x;
        fx0=fx1;
        i = i + 1;
end
fprintf('\nIl metodo non converge!\nNumero di iterazioni: %d\nNumero Valutazioni: %d \n', i, nVal)
    ;
end
```

Per questo metodo si sfrutta l'iterazione

$$x_{i+1} = \frac{f(x_i)x_{i-1} - f(x_{i-1})x_i}{f(x_i) - f(x_{i-1})} \quad i = 1, 2, ..$$

e alla fine stampa il numero delle iterazioni e delle valutazioni richieste.

Metodo delle corde

```
function [ x ] = corde(x0,f,df, iMax, tol)
% Metodo delle Corde criterio d'arresto errore relativo
% Input:)
% - x0: punto di innesco
% - f: stringa con il nome della funzione che implementa la funzione
% - df: stringa con il nome della funzione che implementa la derivata
%       della funzione
% - iMax: numero massimo di iterazioni prefissate
% - tol: tolleranza
% Output:
% - x: approssimazione della radice
fx1=feval(df,x0);
nVal=1;
i = 1;
while i < iMax
        fx=feval(f,x0);
        nVal=nVal+1;
        x = x0 - fx/fx1;
        fprintf('\nx%d = %.16f ', i, x);
        errOnX = abs(x - x0)/(1 + abs(x));
        if (errOnX <= tol)
                fprintf('\n\nIl metodo converge a %.16f\nNumero di iterazioni: %d\n Numero
                    valutazioni: %d\n\n', x, i, nVal);
                return;
        end
        x0 = x;
        i = i + 1;
end
fprintf('\n\nIl metodo NON converge!\nNumero di iterazioni: %d\nT Numero valutazioni:%d \n', i,
    nVal);
end
```

La derivata prima viene valutata solo una volta.

1.6 Esercizio 6

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

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

per $tol = 10^3, 10^6, 10^9, 10^{12}$, partendo da $x_0 = 0$. Per il metodo di bisezione, utilizzare $[0, 1]$, come intervallo di confidenza iniziale, mentre per il metodo delle secanti utilizzare le approssimazioni iniziali $x_0 = 0$ e $x_1 = 1$. Tabulare i risultati, in modo da confrontare le iterazioni richieste da ciascun metodo. Commentare il relativo costo computazionale.

Soluzione

La funzione $f(x)$ non presenta radice multipla in $x = 0$ è una funzione regolare continua e derivabile dove la derivata prima che vale

$$f^{'}(x) = (sin(x) + 1)$$

Per un confronto fra i metodi implementati nell'esercizio precedente si aggiunge come argomento di ritorno la coppia $i, nVal$ che sono rispettivamente il numero delle iterazioni e numero delle valutazioni richieste per raggiungere l'approssimazione desiderata. Per tabulare i risultati da confrontare, tramite una tolleranza variabile, si modificano i metodi implementati in due aspetti. Per una tabulazione compatta i metodi non stampano i risultati intermedi o finali ed inoltre restituiscono come valore finale il numero di iterazioni e di valutazioni necessarie per l'approssimazione richiesta.
Per stampare i risultati tabulati si fa uso della seguente *function* **e6**

```
function e6()
%  e6()
%Funzione per stampare i risultati per numero di iterazione e valutazione
%delle function bisezione, Newton, corde e secanti secondo una
%tolleranza variabile.
f=@(x) x-cos(x);
df=@(x) (sin(x) +1 );
x0=0;
iMax=500;
a=0;
b=1;
temp=3;
fprintf('\nIl metodo bisezione  || Il metodo Newton  || Il metodo Corde   || Il metodo Secanti');
for i = 1 : 4
        temp=3*i;
        [a1,b1]=bisezione( a, b, f, iMax, 10^(-temp));
        [a2,b2]=newton(x0,f,df, iMax, 10^(-temp));
        [a3,b3]=corde(x0,f,df, iMax, 10^(-temp));
        [a4,b4]=secanti(x0,f,df, iMax, 10^(-temp));
        fprintf('\n%d Iter: %d Valut: %d || Iter: %d Valut: %d || Iter: %d Valut: %d|| Iter: %d
            Valut: %d' ,i, a1, b1,a2,b2,a3,b3,a4,b4);
end
fprintf('\n');
end
```

che stampa su la console il seguente risultato

```
Il metodo bisezione  || Il metodo Newton  || Il metodo Corde   || Il metodo Secanti
1 Iter: 10 Valut: 12 || Iter: 4 Valut: 10 || Iter: 17 Valut: 18|| Iter: 4 Valut: 6
2 Iter: 20 Valut: 22 || Iter: 5 Valut: 12 || Iter: 34 Valut: 35|| Iter: 5 Valut: 7
3 Iter: 30 Valut: 32 || Iter: 5 Valut: 12 || Iter: 52 Valut: 53|| Iter: 6 Valut: 8
4 Iter: 40 Valut: 42 || Iter: 6 Valut: 14 || Iter: 69 Valut: 70|| Iter: 6 Valut: 8
>>
```

Ogni metodo presenta i suoi pro e contro e i requisiti di confronto sono il numero di iterazioni e valutazioni, l'onerosità della valutazione della funzione e la praticità del metodo.
Dalla tabella possiamo dire che il metodo che richiede meno iterazioni e quello di Newton confermando il fato che questo metodo ha una convergenza quadratica, superiore agli altri metodi. Il numero delle interazioni del metodo Newton sono uguali al metodo delle Secanti. Utilizzando un tolleranza più piccola si ha un numero inferiore di iterazioni per il metodo Newton in confronto con il metodo delle seccanti.
In caso che la valutazione della funzione e della sua derivata risultino onerose il metodo delle secanti e preferibile poiché ha un numero di valutazioni inferiore al metodo Newton.
Con il metodo di bisezione con il nuovo requisito di arresto, errore relativo, si presenta migliore in prestazioni in confronto al metodo delle corde richiedendo un numero di iterazioni e valutazioni inferiori ad essa. Con la versione originale il metodo di bisezione presenta un numero delle valutazioni due volte più grande.
Il metodo delle corde presenta poche istruzioni di confronto ed un metodo leggero e con un basso costo computazionale e in caso di funzioni semplici diventa un metodo molto pratico.

1.7 Esercizio 7

Calcolare la molteplicità della radice nulla della funzione

$$f(x) = x^2 tan(x),$$

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.

Soluzione

La funzione $f(x)$ è periodica e si può vedere che il punto $x = 0$ rappresenta una radice per la funzione e inoltre si tratta di una radice multipla.
Derivando troviamo che

$$f'(x) = 2x \tan(x) + x^2 \sec^2(x) = x\left(2\tan(x) + x\sec^2(x)\right)$$

Per determinare esattamente la molteplicità della radice in $x = 0$ si fa uso del *Teorema* 2.2 dove la molteplicità della radice m vale

$$\lim_{i\to\infty} \frac{e_{i+1}}{e_i} = \frac{m-1}{m}$$

Per provare sperimentalmente l'uguaglianza sopra indicata si fa uso di due *function*, **multi** e **mRoot** dove la prima implementa il metodo di Newton ma che restituisce un vettore che contiene gli errori relativi di ogni iterazione del metodo. La *function mRoot* elabora i dati contenuti in questo vettore per verificare che

$$\frac{m-1}{m} = \frac{2}{3}$$

confermando il risultato che la molteplicità della funzione $f(x)$ nella radice $x = 0$ vale $m = 3$.
In seguito la *function* **multi**

```
function [m] = multi(x0, iMax, tolx)
%   [m] = multi(x0, iMax, tolx)
% Metodo Newton con criterio d'arresto sul errore relativo
% Input:
% - x0: punto d'innesco
% - iMax: numero massimo di iterazioni
% - tolx: tolleranza
% Output:
% - m: Vettore che contiene gli errori relativi di f.
f=@(x) x^2*tan(x);
df=@(x) x*(2*tan(x)+x*sec(x)^2);
fx = feval(f, x0);
dfx = feval(df, x0);
x = x0 - (fx / dfx);
%   fprintf('\nx = %.16f  %.16f %.16f' , x , fx ,dfx);
nVal = 2;
i = 1;
e1=1;
e2=1;
while (i < iMax )
        fx = feval(f, x0);
        dfx = feval(df, x0);
        x = x0 - (fx / dfx);
        %fprintf('\nx%d = %.16f ' , i, x);
        nVal = nVal + 2;
        errorX = abs(x - x0)/(1 + abs(x));
        if (errorX <= tolx)
                %fprintf('\nIl metodo converge a %.16f\nNumero di iterazioni: %d Numero di
                    valutazioni: %d\n', x, i,nVal);
                return;
```

```
        end
        %----------Codice che tiene tracia del rapporto della molteplicita
        e2=errorX;  %Errore relativo della nuova approssimazione
        m(i)=e2/e1;  % Diviso per l'errore della vecchia approssimazione
        e1=e2;
        x0 = x;
        i = i + 1;
end
%fprintf('\nIl metodo non converge!\nNumero di iterazioni: %d\n %.16f', i , x );
end
```

La *function* **nRoot**

```
function  mRoot()
%  mRoot()
%Metodo utilizzato per verificare se la funzione f(x)=x^2*tan(x)ha
%molteplicita 3.
limite=2/3;  %La funzione ha molteplicita 3,  (m-1)/m
fprintf('\n Calcolo della molteplicita della funzione f(x)=x^2*tan(x)\n');
a=multi(1,100,eps); %chiamata al metodo newton per il calcolo del vettore degli errori relativi
control= true;
i=1;
while control
        if( abs(a(i)- limite)<= 10^(-10) )
                control = false;
        end
        i=i+1;
        end
        if  ~ control
                fprintf('\nLa molteplicita della funzione f(x)=x^2*tan(x) = \n 3 \n');
        else
                fprintf('\n La molteplicita della funzione f(x)=x^2*tan(x) e diversa da \n 3 \n');
        end
end
```

Avendo verificato che la molteplicità di $f(x)$ in $x = 0$ è $m = 3$, si confrontano i tre metodi richiesti (Newton, Newton modificato, Aitken).
Il confronto si basa su due indicatori che sono il numero di iterazione del metodo e il numero delle valutazioni della funzione utilizzate durante l'esecuzione. Le iterazioni vengono valutate per determinare la velocità del metodo e le valutazioni per determinare l'uso di memoria. Lla *function* ausiliaria **e7** per stampa i risultati richiesti.

```
function e7()
% e7()
%Funzione per stampare e tabulare  i risultati per numero di iterazione e valutazione
%degli metodi di Newton, Newtn modificato e Aitken secondo una
%tolleranza variabile.
x0=1; % Punto di innesco
iMax=500;
fprintf('\nStampa dei risultati per riga per tolx rispettivamente 10^(-3), 10^(-6), 10^(-9)
    ,10^(-12)');
fprintf('\n')
fprintf('\n  Il metodo Newton   | Il metodo Newton  | Il metodo Aitken');
temp=3;
for i = 1 : 4
        [a1,b1]=newton2( x0,  iMax, 10^(-i*temp));
        [a2,b2]=newton3(x0, iMax, 10^(-i*temp));
        [a3,b3]=aitken2(x0, iMax, 10^(-i*temp));
        fprintf('\n%d Iter: %d Valut: %d | Iter: %d Valut: %d  | Iter: %d Valut: %d',i, a1, b1,a2,
            b2,a3,b3);
end
fprintf('\n');
end
```

La *function* con punto di innesco $x_0 = 1$ stampa su lo schermo il seguente risultato.

```
Stampa dei risultati per riga per tolx rispettivamente 10^(-3), 10^(-6), 10^(-9),10^(-12)
Il metodo Newton      | Il metodo Newton   | Il metodo Aitken
1 Iter: 17 Valut: 34  | Iter: 4 Valut: 8   | Iter: 4 Valut: 18
2 Iter: 35 Valut: 70  | Iter: 4 Valut: 8   | Iter: 4 Valut: 18
```

```
3 Iter: 52 Valut: 104 | Iter: 5 Valut: 10  | Iter: 5 Valut: 22
4 Iter: 69 Valut: 138 | Iter: 5 Valut: 10  | Iter: 5 Valut: 22
>>
```

Da la prima colona riguardante il metodo Newton si vede che per la radice multipla in $x = 0$, il metodo è mal condizionato confermando la teoria. Infatti il numero di iterazioni e valutazioni richieste è molto grande di conseguenza si può affermare che il metodo Newton in presenza di mal condizionamenti per presenza di radice multipla ha un convergenza lineare.
Dalla tabella si vede che i metodi di Newton modificato e di accelerazione di Aitken hanno una covergenza identica verso l'approssimazione della radice con la differenza che Newton modificato ha un numero di valutazioni inferiore. Il fatto di sapere la molteplicità della radice in qui converge il metodo sicuramente aiuta a costruire un algoritmo più efficiente. Quando non siamo alla conoscenza della molteplicità della radice, il metodo che converge più velocemente è l'accelerazione di Aitken.

In seguito si ha le implementazioni dei metodi richiesti, tenendo presente che nel codice sono presenti le istruzioni per le stampe che per ragioni di riutilizzo dei metodo sono state sotto forma di commento preferendo non eliminare il codice per una migliore compressione. Visto che il metodo di Aitken con l'innesco in $x = 1$ si comporta in modo non prevedibile si esegue un passo del algoritmo di Newton prima di entrare nel ciclo. Per equilibrare i metodi si implementa questa scelta anche per gli altri due metodi. Inoltre vista la convergenza maggiore dei metodi Newton rivisitato e Aitken, per ovviare ai problemi di divisione per zero o presenza di NaN nei calcoli si e introdotto dei controlli dovuti.

Metodo Newton con criterio di arresto basato su tolleranza relativa

```
function [i,nVal ] = newton2(x0, iMax, tolx)
%  [i,nVal ] = newton2(x0, iMax, tolx)
%Metodo NEWTON con criterio d'arresto sul errore relativo
% Input:
% - x0: punto d'innesco
% - iMax: numero massimo di iterazioni
% - tolx: tolleranza
% Output:
% - i: Numero di iterazioni richieste per l'approssimazione del zero
% -nVal: Numero di valutazioni richieste
i = 1;
f=@(x) x^2*tan(x);
df=@(x) x*(2*tan(x)+x*sec(x)^2);
fx = feval(f, x0);
dfx = feval(df, x0);
x = x0 - (fx / dfx);
%   fprintf('\nx = %.16f  %.16f %.16f' , x , fx ,dfx);
nVal = 2;
i=i+1;
while (i < iMax )
        fx = feval(f, x0);
        dfx = feval(df, x0);
        x = x0 - (fx / dfx);
        %fprintf('\nx%d = %.16f ' , i, x);
        nVal = nVal + 2;
        errorX = abs(x - x0)/(1 + abs(x));
        if (errorX <= tolx)
                %fprintf('\nIl metodo converge a %.16f\nNumero di iterazioni: %d Numero di
                    valutazioni: %d\n', x, i,nVal);
                return;
        end
        x0 = x;
        i = i + 1;
end
%fprintf('\nIl metodo non converge!\nNumero di iterazioni: %d\n %.16f', i , x );
end
```

Metodo Newton rivisitato con criterio di arresto basata su tolleranza relativa

```
function [ i,val ] = newton3(x0, max, tolx)
%    [ i,val ] = newton3(x0, max, tolx)
```

```
% Il metodo di Newton modificato con criterio d'arresto del'incremento relativo.
% punto d'innesco in modo tale da determinare la radice 1 del polinomio
% radice 1 e di moltiplicita 3.       m=3.
% Input:
% - x0: punto d'innesco
% - max: numero massimo di iterazioni prefissate
% - tolx: tolleranza usata per il calcolo della soglia d'arresto
% Output:
% - i: Numero di iterazioni richieste per l'approssimazione del zero
% -nVal: Numero di valutazioni richieste
        f=@(x) x^2*tan(x);
        df=@(x) x*(2*tan(x)+x*sec(x)^2);
        fx = feval(f, x0);
        dfx = feval(df, x0);
        val = 2;
        i = 1;
        if (fx == 0)  %se x0 e soluzione della funzione
                x = x0;
                %fprintf('Il metodo converge a %1.16f\nNumero di iterazioni: 1\nNumero di
                    valutazioni di funzioni: %d\n', x, val);
                return;
        end
        if (dfx == 0)  %Se la derivata uguale a zero
                x = x0;
                %fprintf('\nDerivata prima uguale a zero --> impossibile continuare l''iterazione\
                    nApprossimazione della radice ottenuta: %d\nNumero di iterazioni: %d\nNumero
                    di valutazioni di funzioni: %d\n', x, i, val);
                return;
        end
        x1 = x0 - 3*(fx / dfx); %Primo passo di Newton. Molteplicita 3
        %fprintf('\n x1 = %1.16f', x1);
        errorX = abs(x1 - x0)/(1 + abs(x1));
        if (errorX <= tolx)   %approssimazione trovata
                x = x1;
                %fprintf('\nIl metodo converge a %1.16f\nNumero di iterazioni: 1\nNumero di
                    valutazioni di funzioni: %d\n', x, val);
                return;
        end
        i=i+1;
        while (i < max)
                fx = feval(f, x1);
                val = val + 1;
                if(fx==0)
                        %fprintf('\nIl metodo converge a zero a %1.16f\nNumero di iterazioni: %d\
                            nNumero di valutazioni di funzioni: %d\n\n', x,i, val);
                        return;
                end
                dfx = feval(df, x1);
                if (dfx == 0)  %Se la derivata uguale a zero
                        x = x0;
                        %fprintf('\nDerivata prima uguale a zero --> impossibile continuare l''
                            iterazione\nApprossimazione della radice ottenuta: %d\nNumero di
                            iterazioni: %d\nNumero di valutazioni di funzioni: %d\n', x, i, val);
                return;
                end
                val = val + 1;
                x = x1 - 3* (fx / dfx);  %4 molteplicita della radice della funzione
                %fprintf('\n x%d = %.16f ' , i, x);
                errOnX = abs(x - x1)/(1 + abs(x));
                if (errOnX <= tolx ) %controllo dell approssimazion
                        %fprintf('\nIl metodo converge a %1.16f\nNumero di iterazioni: %d\nNumero
                            di valutazioni di funzioni: %d\n\n', x,i, val);
                        return;
                end
                x0=x1;
                x1=x;
                i = i + 1;
        end
        %fprintf('\nIl metodo NON converge! Numero di iterazioni: %d\n', i);
end
```

Metodo Aitke con criterio di arresto basata su tolleranza relativa

```
function [ i,val ] = aitken2(x0, max, tolx)
```

```
%       [ i,val ] = aitken2(x0, max, tolx)
%Il metodo di Newton con accelerazione di Aitken.
% Input:
% - x0: punto d'innesco
% - max: numero massimo di iterazioni prefissate
% - tolx: tolleranza usata per il calcolo della soglia d'arresto
% Output:
% - i: Numero di iterazioni richieste per l'approssimazione del zero
% -nVal: Numero di valutazioni richieste
        f=@(x) x^2*tan(x);
        df=@(x) x*(2*tan(x)+x*sec(x)^2);
        control=false;                      %Controllo della funzione
        x=x0;
        i = 1;
        val=0;
        fx=feval(f,x0);
        dfx=feval(df,x0);
        val=val+2;
        if (fx==0)
                x=x0; %abbiamo trovato la soluzione
                return;
        end
        if(dfx==0)
                %fprintf('Errore la derivala uguale a zero il metodo non converge');
                return;
        end
        x0=x0-fx/dfx; % primo passo del metodo Newton
        while ( i < max )
                fx=feval(f,x0);
                dfx=feval(df,x0);
                val=val+2;
                if (fx==0)
                        x=x0;
                        control = true; %abbiamo trovato la soluzione
                        break;
                end
                if(dfx==0)
                        %fprintf('Errore la derivala uguale a zero il metodo non converge');
                        return;
                end
                x1=x0-fx/dfx; % primo passo del metodo Newton
                fx=feval(f,x1);
                dfx=feval(df,x1);
                val=val+2;
                if (fx==0)
                        x=x1;
                        control = true; %abbiamo trovato la soluzione
                        break;
                end
                if(dfx==0)
                        %fprintf('Errore la derivala uguale a zero il metodo non converge');
                        return;
                end
                x2=x1-fx/dfx; % secondo passo del metodo Newton
                y=x2-2*x1+x0; %controlliamo se denominatore diverso da 0
                if(y==0)
                        %fprinf('Denominatore del equazione di Aitken uguale a zero metodo non
                            utilizabile');
                x=x2;
                return;
                end
                x=(x2*x0-x1.^2)/y;  %accelerazione di Aitken
                %fprintf('\n x%d: =  %.16f' , i, x);
                error = abs(x - x0) / (1 + abs(x)); %errore relativo di x
                if (error <= tolx)
                        control = true;
                        break;
                end
                x0 = x;
                i = i + 1;
                end
                if(control) %Metodo convergente
                        %fprintf('\nIl metodo converge con \nx: %.16f \nNumero di iterazioni: %d \
                            nNumero valutazioni: %d', x,i,val);
                else
                        %fprinf('Il metodo non converge. Raggiunto il massimo numero di iterazioni
                            ');
```

```
        end
end
```

1.8 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*.

Soluzione

La seguente *function* **palu** fa si che data la matrice A non singolare lo fattorizza nel prodotto $pA = LU$ dove L è una matrice triangolare inferiore con diagonale unitaria, U una matrice triangolare superiore e p il vettore di permutazione.

Fattorizzazione LU con pivoting parziale

```
function [ A, p] = palu( A )
%   palu( A )
% Fattorizza la matrice A quadrata in LU con il metodo del pivoting
% parziale pA=LU
% Input:
% -A: matrice nonsingolare.
% Output:
% -A: la matrice fattorizzata L ed U con pivoting parziale;
% -p: vettore di permutazione
[m,n] = size(A);
if m~=n
        error('La matrice non e quadrata');
end
p = 1 : n;
for i = 1 : n - 1
        [mi, ki] = max(abs(A(i : n, i)));
        if mi == 0
                error('La matrice e singolare'); %Se la matrice e singolare termina
        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
end
```

La strategia adottata è quella del pivoting parziale in qui si sceglie come pivot l'elemento più grande in valore assoluto scambiando le righe in modo tale da portare l'elemento scelto nella posizione di pivot. La *funtion* restituisce la matrice A fattorizzata $pA = LU$ e il vettore p delle permutazioni.

1.9 Esercizio 9

Scrivere una *function* Matlab che, data in ingresso la matrice LU ed il vettore p creati dalla *function* del precedente esercizio, ed il termine noto del sistema lineare $Ax = b$, ne calcoli la soluzione:
$function \quad x = lusolve(LU, p, b)$
Curare particolarmente la scrittura e l'efficienza della *function*.

Soluzione

Per implementare la *function* richiesta si fa uso di queste due semplici *function*.
La seguente *funcion* risolve i sistemi lineari con matrici triangolari inferiori con diagonale unitaria.

```
function [b] = triangolareInferiore(A, b)
%   [b] = triangolareInferiore(A, b)
% Risolve sistema lineare con matrice triangolare inferiore con diagonale unitaria
% Input:
% -A: la matrice dei coefficienti
% -b: vettore termini noti.
% Output:
% -b: vettore delle soluzioni
n = size(A,1);
for j=1:n
        for i = j + 1 : n
        b(i) = b(i) - A(i,j) * b(j);
        end
end
end
```

Inoltre la seguente *funcion* risolve i sistemi lineari con matrici triangolari superiori.

```
function [b] = triangolareSup(A, b)
% [b] = triangolareSup(A, b)
% Risolve sistema lineare con matrice triangolare superiore
% Input:
% -A: la matrice dei coefficienti;
% -b: termini noti.
% Output:
% -b: vettore delle soluzioni.
n = size(A,1);
for i = n : -1 : 1
        for j = i+1 : n
                b(i) = b(i) - A(i,j) * b(j);
        end
b(i)=b(i)/A(i,i);
end
end
```

Invece la seguente *funcion* **lusolve** trova le soluzioni del sistema lineare usando la fattorizzazione tramite pivoting parziale. Prende come input la matrice fattorizzata LU, il vettore p di perturbazione e infine il vettore dei termini noto b restituendo la soluzione del sistema.

```
function [x] = lusolve(LU,p,b)
%  [x] = lusolve(LU,p,b)
%Soluzione sistema lineare utilizando LU con pivoting parziale
% Input:
% -LU: matrice nonsingolare fatorizzata LU.
% -p: vettore delle permutazioni
% -b: vettore dei termini noti.
% Output:
% -x: vettore delle soluzioni.
P=zeros(length(LU));
for i=1:length(LU) %Creazione della matrice di permutazione
        P(i, p(i)) = 1;
end
%tril(A,-1) La matrice triandolare inferiore senza la diadonale di A
b = triangolareInferiore(tril(LU,-1)+eye(length(LU)), P*b); %
x = triangolareSup(triu(LU), b); % triu La parte triangolare superiore con diagonale di A
end
```

Utilizzando il comandi Matlab *tril* e *triu* siamo in grado di selezionare porzioni della matrice LU.

1.10 Esercizio 10

Scaricare la *function cremat* al sito:
`http://web.math.unifi.it/users/brugnano/appoggio/linsis.m`
che crea sistemi lineari $n \times n$ la cui soluzione è il vettore $x = (1...n)^T$.
Eseguire, quindi, lo *script* Matlab:

```
n = 10;
xref = (1:10)';
for i = 1:10
        [A,b] = linsis(n,i);
        [LU,p] = palu(A);
        x = lusolve(LU,p,b);
        disp(norm(x-xref))
end
```

Tabulare in modo efficace, e spiegare in modo esauriente, i risultati ottenuti.

Soluzione

Per tabulare in modo efficace i risultati ottenuti e per capire meglio le cifre ottenute si modifica lo script

Script per una migliore tabulazione e compresone dei risultati.

```
%  script
%Script per tabullare i risultati del esercizio 10
clc;
n = 10;
xref = (1:10)';
fprintf('\nTabulazione dei risultati dello script ');
fprintf('\n   Prova numero | Condizionamento della matrice A | Distanza dei vettori  norm(x-xref)|
      ');
for i = 1:10
        [A,b] = linsis(n,i);
        [LU,p] = palu(A);
        x = lusolve(LU,p,b);
        a2=cond(A);
        a1=norm(x-xref);
        fprintf('\n       %d        |           %0.5e           |      %0.5e ' ,i,a2,a1);
end
fprintf('\n');
```

che stampa il seguente risultato

```
Tabulazione dei risultati dello script
Prova numero      | Condizionamento della matrice A | Distanza dei vettori  norm(x-xref)|
         1        |              1.00000e+01        |       1.98593e-14
         2        |              5.00000e+01        |       1.26487e-14
         3        |              5.00000e+03        |       5.27154e-13
         4        |              5.00000e+05        |       6.37276e-11
         5        |              5.00000e+07        |       1.20909e-08
         6        |              5.00000e+09        |       1.44717e-06
         7        |              4.99996e+11        |       1.36783e-04
         8        |              5.00150e+13        |       1.00581e-02
         9        |              5.09977e+15        |       6.20152e-01
         10       |              7.11979e+18        |       1.32667e+03
>>
```

Lo script crea dieci diverse matrici A con i rispettivi termini noti b per poi chiamare i metodi implementati per la risoluzione dei sistemi lineari tramite il metodo di eliminazione di Gauss con pivoting parziale, conoscendo la soluzione delle equazioni, $x = [1, 2, ..., n]^{'}$.
Le matrici create sono via via sempre più mal condizionate come si vede anche dalla seconda colonna della tabulazione. Lo script vuol far vedere che in presenza di matrici mal condizionate la soluzione dei sistemi con il metodo scelto, la soluzione distanzia dalla soluzione esatta. Infatti, più la matrice A è mal condizionata, più cresce la distanza della soluzione dal risultato esatto. Concludendo, il metodo di fattorizzazione LU di eliminazione di Gauss con pivoting parziale presenta delle problematiche in presenza di matrici mal

condizionate, più grande è il mal condizionamento della matrice, più si distanzia dalla soluzione esatta. Il metodo è sensibile in presenza di matrici mal condizionate. Invece in presenza di matrici ben condizionate il metodo è ben condizionato ed efficiente.

1.11 Esercizio 11

Scrivere una *function* Matlab che, data in ingresso una matrice $A \in \mathcal{R}^{m \times n}$ 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*.

Soluzione

La seguente *function* **myqr** prende come input la matrice A ed effettua la decomposizione $A = QR$ dove la parte superiore di QR contiene la matrice triangolare superiore R mentre la parte strettamente inferiore conterà i vettori di Householder.

```
function [A] = myqr(A)
%       [A] = myqr(A)
% Questa funzione definisce una procedura per fattorizzare QR una matrice
% A data in input. Restituisce la matrice stessa fattorizzata.
% Input:
% - A: matrice dei coefficienti
% Output:
% - A: la parte superiore di QR contiene la matrice triangolare
% superiore R mentre la parte strettamente inferiore contera i vettori di Householder
[m,n] = size(A);
for i=1:n
        alpha = norm(A(i:m, i));
        if alpha==0
                error('La matrice A non ha rango massimo');
        end
        if A(i,i)>=0
        alpha = -alpha;
        end
        v = A(i,i) - alpha;
        A(i,i) = alpha;
        A(i+1:m,i) = A(i+1:m,i)/v;
        beta = -v/alpha;
        A(i:m,i+1:n) = A(i:m, i+1:n)- (beta*[1; A(i+1:m,i)])*([1 A(i+1:m,i)']*A(i:m,i+1:n));
end
end
```

Per prima cosa si controlla che la matrice ha rango massimo per poi procedere nella fattorizzazione QR. Il metodo consiste nella costruzione delle particolari matrici ortogonali $Hz = \alpha e_1$ dove $\alpha = \pm||z||_2$ dette matrici di Householder che definiscono il vettore di Householder $v_1 = z_1 - \alpha$. La scelta di segno di α è importante per rendere sempre ben condizionata la costruzione del vettore v di Householde. Infatti z_1 e α devono essere di segno opposto per avere una buon condizionamento nella somma algebrica.

1.12 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 $Ax = b$, ne calcoli la soluzione nel senso dei minimi quadrati:
function $x = qrsolve(QR, b)$
Curare particolarmente la scrittura e l'efficienza della *function*.

Soluzione

La seguente *function* **qrsolve** risolve il sistema lineare dove prende come input la matrice dei coefficienti QR fattorizzata QR e il vettore dei termini noti b restituendo la soluzione del sistema.

```
function [x] = qrsolve(QR,b)
%    [x] = qrsolve(QR,b)
% % Risoluzione del sistema lineare sovradeterminato fattorizzato QR.
% Input:
% - QR: matrice dei coefficienti fattorizzato QR
% -b: vettore termini noti.
% Output:
% - x : vettore delle soluzioni.
[m,n] = size(QR);
for i =1:n
        v =[1;QR(i+1:m,i)];
        b(i:m)= b(i:m)-((2*v*(v'))/((v') * v))* b(i:m);
end
x=triangolareSup(QR(1:n,1:n),b(1:n));
end
```

1.13 Esercizio 13

Utilizzare le *function* scritte negli esercizi 11 e 12 per risolvere, nel senso dei minimi quadrati, il sistema lineare sovradeterminato definito dai seguenti dati:

$A =$

$$\begin{matrix} 1 & 2 & 3 \\ 1 & 2 & 4 \\ 3 & 4 & 5 \\ 3 & 4 & 6 \\ 5 & 6 & 7 \end{matrix}$$

$b =$

$$\begin{matrix} 14 \\ 17 \\ 26 \\ 29 \\ 38 \end{matrix}$$

Soluzione

Per la soluzione del esercizio si fa uso dello script seguente.

```
function [x] = e13()
%       e13()
%Funzione per la soluzione del esercizio 13
% Output:
% -x: vettore delle soluzioni.
disp(' A: matrice nonsingolare dei coeffienti');
A=[1 2 3;1 2 4;3 4 5;3 4 6;5 6 7];
A
disp(' b: vettore dei termini noti.');
b=[14;17;26;29;38];
b
A = myqr(A);
x=qrsolve(A,b);
disp('Solozione del sistema lienare tramite la fattorizzazione QR ');
end
```

che stampa sul terminale la soluzione del sistema tramite il metodo di fattorizzazione QR tramite i vettori di Householder.

```
ans =
1.0000
2.0000
3.0000
>>
```

1.14 Esercizio 14

Le seguenti istruzioni,

```
A = rot90(vander(1:10)); A = A(:,1:8); x = (1:8)'; b = A*x;
```

creano una matrice, $A \in \mathcal{R}^{10\times 8}$ di rango massimo ed un vettore $b \in \mathcal{R}^8$, che definisce un sistema lineare la cui soluzione è data dal vettore

$$x = (1, 2, 3, 4, 5, 6, 7, 8)^T$$

Spiegare quindi qual è il significato delle espressioni Matlab:

```
A\b,                    (A'*A)\(A'*b)
```

Spiegare i risultati ottenuti.

Soluzione

Per prima cosa le istruzioni creano una matrice di Vandermonde e lo memorizza nella variabile A. Una delle proprietà di questa matrice è che hanno rango massimo sono non singolare e sono malcondizionate infatti A presenta un condizionamento del ordine di $1e^9$. In seguito il codice ridimensiona la matrice che ora diventa 10×8.
Definendo le variabili x e b si cerca in due modi differenti di trovare la soluzione del sistema lineare $Ax = b$. Per la proprietà della matrice trasposta i due metodi sono equivalenti, infatti vale l'equivalenza

$$A \setminus b \equiv (A'A) \setminus (A'b)$$

I due metodi usano lo stesso commando Matlab, *mldivide* però usano algoritmi differenti. Per la soluzione del sistema lineare del primo metodo si utilizza la fattorizzazione QR del sistema sovradeterminato. Per il secondo metodo si usa la decomposizione di Cholesky per la soluzione del sistema lineare. Il prodotto $A^{'} * A = B$ produce una matrice sdp, simmetrica e definita positiva. Infatti la matrice B è simmetrica e vale la proprietà che per ogni $x \in \mathbb{R}, x \neq 0$ risulta $x^{'}Bx > 0$. La matrice B è malcondizionata questo perchè il prodotto $A'A$ crea una matrice malcondizionata del ordine di $1e^{18}$. Questo risultato viene confermato anche con l'avviso del programma che mette in guardia per risultati inaccurati perché si utilizza una matrice malcondizionata. Avviso per altro veritiero, i risultati del secondo metodo sono discostanti da i risultati aspettati. Il primo metodo si comporta meglio e considerando il malcondizionamento della matrice A ottiene dei risultati accettabili questo anche del fatto che la fattorizzazione QR si comporta meglio in presenza di matrici malcondizionate. Il secondo metodo anche se più semplice da dei risultati discostanti e per questo non è preferibile in presenza di matrici malcondizionate.

1.15 Esercizio 15

Approssimare la funzione $f(x) = cos(\pi x^2/2)$ con i polinomi interpolanti rispettivamente costruiti con $n+1$ ascisse equidistanti e con $n+1$ ascisse di Chebyshev sull'intervallo $[-1; 1]$. Graficare (in formato semilogy) il massimo errore di interpolazione, per $n = 1, 2, ..., 40$. Commentare i risultati ottenuti.

Soluzione

Per la risoluzione del problema si é scelto il metodo di Newton per determinare il polinomio interpolante. Considerando il $Teorema$ 4.1 sul esistenza e l'unicità del polinomio interpolante, anche il metodo di Lagrange è equivalente.
Il polinomio interpolante di Newton si genera in modo ricorsivo dove per base di Newton si ha i seguenti dati

$$w_0(x) = 1$$
$$w_{k+1} = (x - x_k)w_k(x)$$

dove $w_k(x) = (x - x_0)(x - x_1)....(x - x_n)$ per $k = 0, 1, 2, ...n$
Per scrivere il polinomio di interpolazione si ha bisogno delle quantità dette differenze divise. Una differenza divisa di primo ordine si presenta come

$$f[x_0, x_1] = \frac{f[x_1] - f[x_0]}{x_1 - x_0}$$

Una differenza divisa di secondo ordine si presenta come

$$f[x_0, x_1, x_2] = \frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0}$$

Una differenza divisa di ordine n ha la seguente forma

$$f[x_0, x_1, x_2...x_n] = \frac{f[x_1, x_2...x_n] - f[x_0, x_1...x_{n-1}]}{x_n - x_0}$$

Insieme costituiscono la tabella delle differenze divise dove gli elementi della diagonale vengono utilizzati per la costruzione del polinomio di interpolazione di Newton.

$$p_r(x) = f[x_0] + f[x_0, x_1](x - x_0) + ... + f[x_0, x_1....x_r](x - x_0)(x - x_1).....(x - x_{r-1})$$

La seguente function implementa il calcolo del polinomio interpolante di Newton.

```
function [y,dfi] = newton( xi, fi, x )
% [y,dfi] = newton( xi, fi, x ) Calcola il valore del polinomio
% interpolante i punti (xi,fi) nei punti del vettore x secondo il metodo Newton.
% - xi: ascisse interpolanti
% - fi: il valore della funzione delle ascisse interpolanti
% - x: punti da valutare
% Output:
% - y:  Le coordinate dei punti x
% - dfi. Le differenze divise
        n = length(xi)-1; % grado del polinomio interpolante
        if n~=length(fi)-1, error('dati inconsistenti')
        else
        for i = 1:n
                if any( find(xi(i+1:n)==xi(i)) )
                        error('ascisse non distinte'), end
                end
        end
        dfi = fi;
        for i = 1:n
                for j = n+1:-1:i+1
                        dfi(j) = ( dfi(j)-dfi(j-1) )/( xi(j)-xi(j-i) );
                end
        end
        y = dfi(n+1)*ones(size(x)); %Horner generalizato
        for k = 0:n-1
                y = y.*( x-xi(n-k) ) + dfi(n-k);
        end
        return
end
```

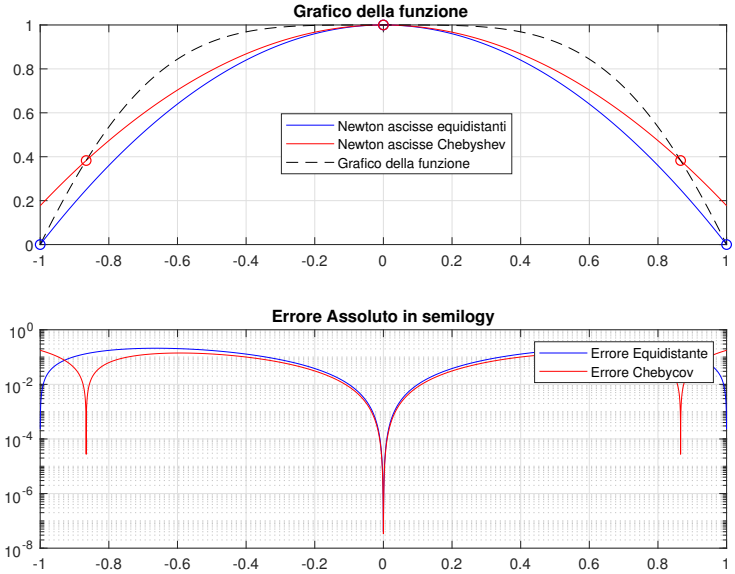
Per costruire le scisse di Chebyshev si usa la seguente *function*.

```
function [xi] = ceby(n,a,b)
%         [xi] = ceby(n,a,b)
% Function per la determinazione delle ascisse di Chebyshev
% - n: Grado del polinomio interpolante
% - a: estremo sinistro del intervallo interpolante [a,b]
% - b: estremo destro del intervallo interpolante [a,b]
% Output:
% - xi:  vettore contenete le ascisse di Chebyshev di lunghezza n+1.
        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
```

Per graficare le differenze della scelta delle ascisse rispettivamente ascisse equidistanti e ascisse di Chebyshev si fa uso della funzione $e15$ che prende come argomento d'input il grado del polinomio desiderato.

```
function e15(n)
%   e15(n)  Funzione per calcolare i risultati del esercizio 15
%   Mostra e confronta i grafici delle funzioni costruite ripetivamente con le
%   asscisse equidistanti e le ascisse di Chebyshev.
%   % - xi: asisse interpolanti
        f=@(x) cos((pi*x.^2)/2);
        a=-1;
        b=1;
        xi=linspace(a,b,n+1);
        fi=f(xi);
        k=10001;  %punti da valutare meglio dispari
        x=linspace(a,b,k);
        fx=f(x);
        y=newton(xi,fi,x);
        xi2=ceby(n,a,b);
        fi2=f(xi2);
        y2=newton(xi2,fi2,x);
        figure
        subplot(2 ,1 ,1);
        plot(x,y,'b',x,y2,'r',x,fx,'k--',xi,fi,'bo',xi2,fi2,'ro');
        title ('Grafico della funzione ');
        grid on;
        legend('Newton ascisse equidistanti','Newton ascisse Chebyshev','Grafico della funzione');
        e=abs(y-fx);
        e1=abs(y2-fx);
        subplot(2,1,2);
        semilogy(x,e,'b',x,e1,'r');
        title ('Errore Assoluto in semilogy')
        grid on
        legend('Errore Equidistante','Errore Chebycov');
        norm(e)
        norm(e1)
        end
```

La funzione costruisce due insiemi di grafi dove nel primo insieme mette a confronto i grafi costruiti secondo le ascisse scelte del metodo Newton e nel secondo insieme grafica gli errori assoluti tramite il comando *semilogy*. Nella seguente figura si mettono in confronto i grafi di grado due che contengono esattamente 3 ascisse.



Invece se si desidera confrontare visivamente la norma dei errori assoluti per tutti i gradi del polinomio interpolante con ascisse differenti si fa uso della funzione *stampa*.

```
function stampa()
```

```
%   stampa()  Stampa i risultati per l'esercizio 15
%Stampa in linea di commando la norma del errore assoluto
%rispetivamente al metodo Newton con ascisse equidistanti
% e i metodo Newton con ascisse di Chebyshev
f=@(x) cos((pi*x.^2)/2);
        a=-1;
        b=1;
        k=101;  %punti da valutare meglio dispari
        x=linspace(a,b,k);
        fprintf('\nN    ||Norma errore equidistanti|| Norma errore Newton ascisse Chebyshev');
        for n=1 : 40
                xi=linspace(a,b,n+1);
                fi=f(xi);
                fx=f(x);
                y=newton(xi,fi,x);
                xi2=ceby(n,a,b);
                fi2=f(xi2);
                y2=newton(xi2,fi2,x);
                e=norm(abs(y-fx));
                e1=norm(abs(y2-fx));
                fprintf('\n%d    ||        %d      ||            %d' ,n, e, e1);
                end
        end
```

che stampa il seguente risultato

N	||Norma errore equidistanti||	Norma errore Newton ascisse Chebyshev
1	|| 8.288494e+00 ||	2.986699e+00
2	|| 1.347341e+00 ||	9.302457e−01
3	|| 9.869864e−01 ||	8.068421e−01
4	|| 1.912131e−01 ||	1.088933e−01
5	|| 1.432603e−01 ||	9.285309e−02
6	|| 2.785375e−02 ||	1.194044e−02
7	|| 1.837636e−02 ||	1.023338e−02
8	|| 2.897777e−03 ||	8.849485e−04
9	|| 2.115229e−03 ||	7.429983e−04
10	|| 3.693219e−04 ||	6.193790e−05
11	|| 2.494943e−04 ||	5.245848e−05
12	|| 2.863499e−05 ||	3.341717e−06
13	|| 2.093978e−05 ||	2.782603e−06
14	|| 3.185933e−06 ||	1.717045e−07
15	|| 2.194721e−06 ||	1.442836e−07
16	|| 1.923972e−07 ||	7.234932e−09
17	|| 1.409580e−07 ||	6.009172e−09
18	|| 1.880337e−08 ||	2.924585e−10
19	|| 1.310924e−08 ||	2.460036e−10
20	|| 9.209303e−10 ||	1.002334e−11
21	|| 6.751090e−10 ||	8.407147e−12
22	|| 7.985306e−11 ||	3.436202e−13
23	|| 5.624465e−11 ||	2.918713e−13
24	|| 3.581973e−12 ||	5.417935e−14
25	|| 1.665425e−12 ||	4.483842e−14
26	|| 5.875186e−12 ||	8.560533e−14
27	|| 9.318950e−12 ||	7.693755e−14
28	|| 4.082810e−11 ||	7.583253e−14
29	|| 6.222755e−12 ||	6.376313e−14
30	|| 1.042983e−10 ||	8.913485e−14
31	|| 1.935531e−11 ||	8.266521e−14
32	|| 5.915222e−10 ||	8.824608e−14
33	|| 7.290158e−11 ||	1.241572e−13

34	||	2.637960e−09	||	7.838743e−14
35	||	4.962760e−10	||	1.334879e−13
36	||	2.563288e−09	||	1.212354e−13
37	||	9.733409e−10	||	1.106354e−13
38	||	2.671530e−08	||	2.312778e−13
39	||	1.348656e−09	||	1.199382e−13
40	||	5.340786e−08	||	1.283641e−13

Definendo p^* il polinomio di miglior approssimazione di grado n della funzione $f(x)$, si ha un errore di interpolazione

$$||e|| \leq (1 + \Lambda_n)||f - p^*||$$

dove Λ_n è la costante di Lebesgue e il numero di condizionamento del problemae che ha le seguenti proprietà.

- Non dipende dal intervallo $[a, b]$ di interpolazione
- La sua crescita dipende dalla distribuzione delle ascisse di interpolazione
  - In caso di ascisse equidistanti cresce esponenzialmente rispetto a n.
  - In caso delle ascisse di Chebyshev ha una crescita logaritmica rispetto a n

Dai risultati ottenuti si può affermare che utilizzando il polinomio interpolante con base di Newton la scelta delle ascisse di Chebyshev è la scelta migliore poiché si commette un errore massimo più piccolo in confronto alle ascisse equidistanti. Interessante è il comportamento del metodo Newton quando il grado del polinomio interpolante è maggiore di 30 cioè il problema di interpolazione comincia a diventare mal condizionato. In questo caso la qualità dell'approssimazione peggiora, ma anche qui la scelta delle ascisse di Chebyshev risolve meglio il problema di minimassimo.

1.16 Esercizio 16

Approssimare la funzione $f(x) = cos(\pi x^2/2)$ con i polinomi interpolanti di Hermite rispettivamente costruiti con $n + 1$ ascisse equidistanti e con $n + 1$ ascisse di Chebyshev sull'intervallo $[-1; 1]$. Graficare (in formato semilogy) il massimo errore di interpolazione, per $n = 1, 2, ..., 40$. Commentare i risultati ottenuti.

Soluzione

L'interpolazione di Hermite è simile alla interpolazione di Newton con la differenza che siamo in possesso delle coordinate della derivata delle ascisse di interpolazione che si utilizzano per il calcolo. Quando nella interpolazione di Newton sono note la coppia dei dati $(x, f(x))$ nella interpolazione di Hermite si usano i seguenti dati $(x, f(x), f'(x))$.
Il polinomio di Hermite ha la seguente forma

$$p(x) = f[x_0] + f[x_0, x_0](x - x_0) + f[x_0, x_0, x_1](x - x_0)^2 + f[x_0, x_0, x_1](x - x_0)^2(x - x_1)....$$

La seguente function implementa il calcolo del polinomio interpolante di Hermite.

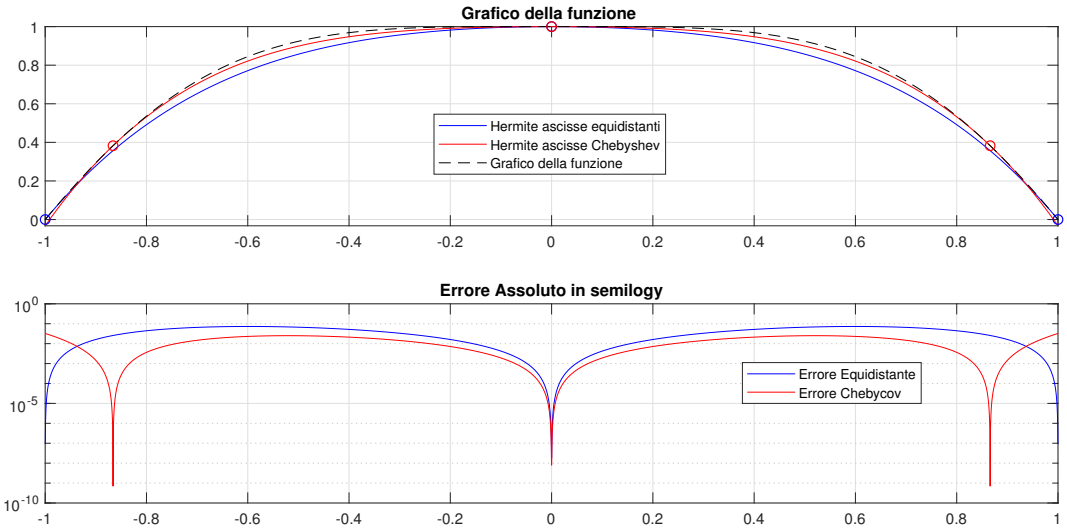
```
function [y,df] = hermite( xi, fi, f1i, xx )
% [y,dfi] = hermite( xi, fi,fli, x ) Calcola il valore del polinomio
% interpolante i punti (xi,fi) nei punti del vettore x secondo il metodo di Hermite.
% - xi: asisse interpolanti
% - fi: il valore della funzione delle ascisse interpolanti
% - fli: il valore della derivata della funzione delle ascisse interpolanti
% - x: punti da valutare
% Output:
% - y:  Le coordinate dei punti x
% - dfi. Le differenze divise
        m = length(xi);
        if m~=length(fi) || m~=length(f1i), error('dati inconsistenti'), end
```

```
        for i = 1:m-1
                if any( find(xi(i+1:m)==xi(i)) ), error('ascisse non distinte'), end
        end
        n = 2*m-1; % grado del polinomio interpolante
        x = zeros(n+1,1);
        df = x;
        x(1:2:n) = xi(:);
        x(2:2:n+1) = xi(:);
        df(1:2:n) = fi(:);
        df(2:2:n+1) = f1i(:);
        for i = n:-2:3 % seconda colonna della tabella
                df(i) = ( df(i)-df(i-2) )/( x(i)-x(i-1) );
        end
        for i = 2:n % colonne successive della tabella
                for j = n+1:-1:i+1
                        df(j) = ( df(j)-df(j-1) )/( x(j)-x(j-i) );
                end
        end
        y = df(n+1)*ones(size(xx));
        for k = 0:n-1
                y = y.*( xx-x(n-k) ) +df(n-k);
        end
        return
end
```

Per costruire le ascisse di Chebyshev si usa la funzione *ceby* dell'esercizio precedente. Per graficare le differenze della scelta delle ascisse rispettivamente ascisse equidistanti e ascisse di Chebyshev si fa uso della funzione $e16$ indicando con n il numero delle ascisse interpolanti la funzione prende come input il numero $n-1$.

```
function e16(n)
%   e16(n)  Funzione per calcolare i risultati del esercizio 16
%   Mostra e confronta i grafici delle funzioni costruite ripetivamente con le
%   asscisse equidistanti e le ascisse di Chebyshev.
%   % - xi: asisse interpolanti
        f=@(x) cos((pi*x.^2)/2);
        df=@(x) (-pi*x.*sin((pi*x.^2)/2));
        a=-1;
        b=1;
        xi=linspace(a,b,n+1);
        fi=f(xi);
        fli=df(xi);
        k=10001;  %punti da valutare meglio dispari
        x=linspace(a,b,k);
        fx=f(x);
        y=hermite(xi,fi,fli,x);
        xi2=ceby(n,a,b);
        fi2=f(xi2);
        fli2=df(xi2);
        y2=hermite(xi2,fi2,fli2,x);
        figure
        subplot(2,1,1);
        plot(x,y,'b',x,y2,'r',x,fx,'k--',xi,fi,'bo',xi2,fi2,'ro');
        title ('Grafico della funzione ');
        grid on;
        legend('Hermite ascisse equidistanti','Hermite ascisse Chebyshev','Grafico della funzione'
            );
        e=abs(y-fx);
        e1=abs(y2-fx);
        subplot(2,1,2);
        semilogy(x,e,'b',x,e1,'r');
        title ('Errore Assoluto in semilogy')
        grid on
        legend('Errore Equidistante','Errore Chebycov');
        norm(e)
        norm(e1)
end
```

La funzione costruisce due insiemi di grafi dove nel primo insieme mette a confronto i grafi costruiti secondo le ascisse scelte dal metodo Hermite e nel secondo insieme grafica gli errori assoluti tramite il comando *semilogy*. Nella seguente figura si mettono a confronto i grafi di grado due che contengono esattamente 3 ascisse.



Invece se si desidera confrontare visivamente la norma dei errori assoluti per tutti i gradi del polinomio interpolante con ascisse differenti si fa uso della funzione *stampa*.

```
function stampa()
%   stampa()  Stampa i risultati per l'esercizio 15
%Stampa in linea di commando la norma del errore assoluto
%rispetivamente al metodo Hermite con ascisse equidistanti
% e i metodo Newton con ascisse di Chebyshev
        f=@(x) cos((pi*x.^2)/2);
        df=@(x) (-pi*x.*sin((pi*x.^2)/2));
        a=-1;
        b=1;
        k=101;  %punti da valutare meglio dispari
        x=linspace(a,b,k);
        fprintf('\nN    ||Norma errore equidistanti|| Norma errore Hermite ascisse Chebyshev');
        for n=1 : 20
                xi=linspace(a,b,n+1);
                fi=f(xi);
                fli=df(xi);
                fx=f(x);
                y=hermite(xi,fi,fli,x);
                xi2=ceby(n,a,b);
                fi2=f(xi2);
                fli2=df(xi2);
                y2=hermite(xi2,fi2,fli2,x);
                e=norm(abs(y-fx));
                e1=norm(abs(y2-fx));
                fprintf('\n%d    ||        %d      ||             %d' ,n, e, e1);
        end
end
```

che stampa il seguente risultato

N	Norma errore equidistanti	Norma errore Hermite ascisse Chebyshev
1	3.384851e+00	1.448906e+00
2	4.512179e−01	1.567810e−01
3	3.197115e−02	1.765343e−02
4	3.141881e−03	1.279078e−03
5	2.581389e−04	9.066909e−05
6	2.055012e−05	4.808276e−06
7	1.665302e−06	2.498490e−07
8	1.052178e−07	1.043647e−08
9	8.018170e−09	4.281336e−10
10	4.134320e−10	1.472458e−11
11	2.925680e−11	4.934134e−13

12	||	1.324112e−12	||	7.352870e−14
13	||	3.057714e−13	||	4.977168e−14
14	||	1.610931e−12	||	6.557700e−14
15	||	4.812819e−13	||	7.571304e−14
16	||	9.838570e−13	||	1.178081e−13
17	||	5.429376e−11	||	6.407106e−14
18	||	7.393861e−11	||	6.566931e−14
19	||	1.068677e−10	||	1.192642e−13
20	||	3.753093e−10	||	8.872923e−14

Definendo p^* il polinomio di miglior approssimazione di grado n della funzione $f(x)$, si ha un errore di interpolazione

$$||e|| \leq (1 + \Lambda_n)||f - p^*||$$

dove Λ_n è la costante di Lebesgue e il numero di condizionamento del problema e che ha le seguenti proprietà.

- Non dipende dal intervallo $[a, b]$ di interpolazione
- La sua crescita dipende dalla distribuzione delle ascisse di interpolazione
  - In caso di ascisse equidistanti cresce esponenzialmente rispetto a n.
  - In caso delle ascisse di Chebyshev ha una crescita logaritmica rispetto a n

Da i risultati ottenuti si può constatare che utilizzando il polinomio interpolante con base di Hermite la scelta delle ascisse di Chebyshev è la scelta migliore poiché si commette un errore massimo più piccolo in confronto alle ascisse equidistanti. Interessante è il comportamento del metodo Hermite quando il grado del polinomio interpolante è maggiore di 12 cioè il problema di interpolazione comincia a diventare mal condizionato. In questo caso la qualità dell'approssimazione peggiora, ma anche qui la scelta delle ascisse di Chebyshev si comporta meglio, presenta una miglior approssimazione.

1.17 Esercizio 17

Utilizzando la function *spline*0 vista in esercitazione, costruire una function Matlab, **splinenat**, avente la stessa sintassi dell function **spline**, che calcoli la spline cubica naturale interpolante una funzione.

Soluzione

Per la determinazione degli n polinomi a tratti che formano una spline cubica si usa la seguente funtion *splineCubica*. Per prima cosa si richiama la *function spline*0 per determinare i momenti m_i della spline naturale per poi calcolare il vettore risultante che contiene i polinomi a tratti che determina la spline cubica naturale.

```
function [ p ] = splinecubica(xi, fi)
% [ p ] = splinecubica(xi, fi) Determina degli n polinomi che formano una spline
% cubica di tipo naturale.
% Input:
% - xi: vettore delle ascisse d'interpolazione
% - fi: vettore dei valori assunti nelle ascisse d'interpolazione
% Output:
% - p: il vettore contenente le n espressioni dei polinomi
        m = spline0(xi,fi);
        ni = length(xi) - 1;
        p = sym('x' , [ni 1]); %Crea un vettore di variabili simboliche
        syms x;
        for i = 2 : ni + 1
                hi = xi(i) - xi(i - 1);
                ri = fi(i - 1) - hi^2/6 * m(i - 1);
                qi = (fi(i) - fi(i - 1))/hi - hi/6 * (m(i) - m(i - 1));
                p(i-1)=((x-xi(i-1))^3 *m(i) + (xi(i) - x)^3 *m(i-1))/(6*hi)+qi*(x-xi(i-1))+ ri;
```

```
        end
end
```

La function principale **splinenat** calcola le coordinate tramite spline naturale per le ascisse richiesti. Per prima cosa richiama la *function* precedente per la costruzione del vettore che contiene i polinomi a tratti e per ogni punto d'ascissa determina quale polinomio applicare restituendo il risultato richiesto.

```
function [x] = splinenat(xi, fi, x)
%  [y] = splinenat(xi, fi, x)  Spline cubica naturale per i punti richiesti.
% Input:
% - xi: ascisse di interpolazione
% - fi: valori della funzione nelle ascisse di interpolazione
% - x:  vettore di ascisse
% Output:
% - x: il vettore contenente le coordinate delle ascisse x
        n = length(xi) - 1;
        if length(xi) < 3
                error('Inserire almeno 4 ascisse interpolanti!');
        end
        fi=splinecubica(xi, fi);
        j = 1;
        k = 1;
        for i = 1 : n
                control = true;
                while j <= length(x) && control
                        if x(j) >= xi(i) && x(j) <= xi(i + 1)
                                j = j+1;
                        else
                                control = false;
                        end
                end
                x(k :j -1) = eval(subs(fi(i), x(k : j-1)));
                k = j;
        end
end
```

1.18 Esercizio 18

Utilizzare la function **splinenat** su definita per approssimare la funzione $f(x) = cos(\pi x^2/2)$ rispettivamente costruiti con $n+1$ ascisse equidistanti e con $n+1$ ascisse di Chebyshev sull'intervallo $[-1;1]$. Graficare (in formato semilogy) il massimo errore di interpolazione, per $n = 4, 5, ..., 100$. Commentare i risultati ottenuti.

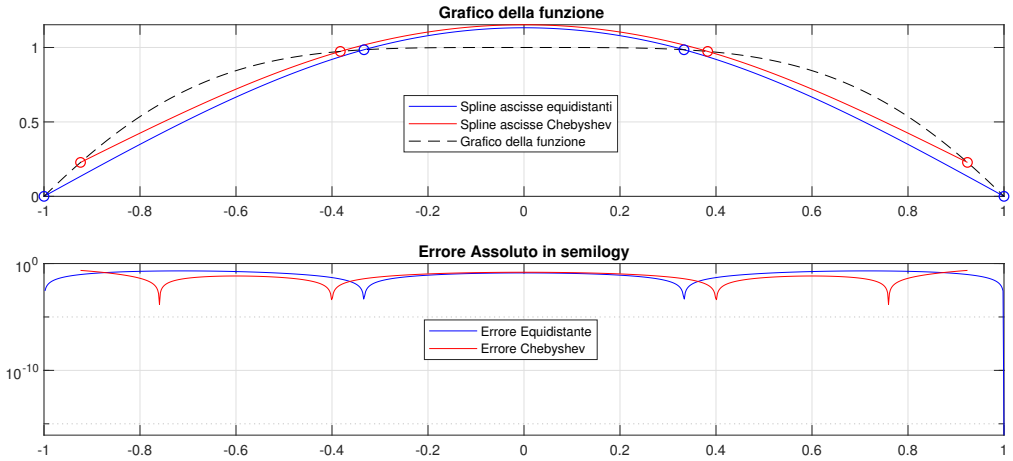
Soluzione

Per costruire le ascisse di Chebyshev si usa la funzione *ceby* dell'esercizio precedente e inoltre i punti di valutazione vengono ridefiniti in modo tale da avere una interpolazione dei punti da valutare. Per graficare le differenze della scelta delle ascisse rispettivamente ascisse equidistanti e ascisse di Chebyshev si fa uso della funzione $e18$ indicando con $n+1$ il numero delle ascisse interpolanti.

```
function e18(n)
%   e18(n)  Funzione per calcolare i risultati del esercizio 18
%   Mostra e confronta i grafici delle funzioni interpolate tramite
%   spline naturale ripetivamente con le
%   asscisse equidistanti e le ascisse di Chebyshev.
%   % - n+1 : asisse interpolanti
        f=@(x) cos((pi*x.^2)/2);
        a=-1;
        b=1;
        xi=linspace(a,b,n+1);
        fi=f(xi);
        k=1001;
        x=linspace(a,b,k);
        fx=f(x);
        y=splinenat(xi,fi,x);
        xi2=ceby(n,a,b)';
        fi2=f(xi2);
        x2=linspace(xi2(1),xi2(end),k); %Ridefiniamo i punti di interpolazione
```

```
        y2=splinenat(xi2,fi2,x2);
        figure
        subplot(2,1,1);
        plot(x,y,'b',x2,y2,'r',x,fx,'k--',xi,fi,'bo',xi2,fi2,'ro');
        title ('Grafico della funzione ');
        grid on;
        legend('Spline ascisse equidistanti','Spline ascisse Chebyshev','Grafico della funzione');
        e=abs(y-fx);
        e1=abs(y2-fx);
        subplot(2,1,2);
        semilogy(x,e,'b',x2,e1,'r');
        title ('Errore Assoluto in semilogy')
        grid on
        legend('Errore Equidistante','Errore Chebyshev');
        norm(e)
        norm(e1)
        end
```

La funzione costruisce due insiemi di grafi dove nel primo insieme mette in confronto i grafi costruiti secondo le ascisse scelte tramite spline naturale e nel secondo insieme grafica le norme degli errori assoluti tramite il comando *semilogy*. Nella seguente figura si mettono a confronto i grafi con 4 ascisse le quali sono anche presentate sul grafo.



Invece se si desidera confrontare visivamente la norma dei errori assoluti per tutti i gradi del polinomio interpolante con ascisse differenti si fa uso della funzione *stampa*18.

```
function stampa18()
%   stampa18()  Stampa i risultati per l'esercizio 18
%Stampa in linea di commando la norma del errore assoluto
%rispetivamente al metodo spline natirale con ascisse equidistanti
% e i metodo spline naturale con ascisse di Chebyshev
        f=@(x) cos((pi*x.^2)/2);
        a=-1;
        b=1;
        k=101;
        x=linspace(a,b,k);
        fprintf('\nN    ||Norma errore equidistanti|| Norma errore Spline ascisse Chebyshev');
        for n=4 : 100
        xi=linspace(a,b,n+1);
        fi=f(xi);
        fx=f(x);
        y=splinenat(xi,fi,x);
        xi2=ceby(n,a,b);
        fi2=f(xi2);
        x2=linspace(xi2(1),xi2(end),k);
        y2=splinenat(xi2,fi2,x2);
        e=norm(abs(y-fx));
        e2=norm(abs(y2-fx));
        fprintf('\n%d    ||          %d      ||              %d' ,n, e, e2);
        end
end
```

che per ovvie questione di spazio presentiamo solo una parte dei risultati.

N	Norma errore equidistanti	Norma errore Spline ascisse Chebyshev
4	4.564064e−01	4.790282e−01
9	3.526117e−02	1.526658e−01
14	9.856669e−03	6.885423e−02
19	4.319046e−03	3.888451e−02
24	2.338011e−03	2.492752e−02
29	1.430157e−03	1.732326e−02
34	9.474594e−04	1.273181e−02
39	6.622891e−04	9.749668e−03
44	4.807734e−04	7.704766e−03
49	3.606105e−04	6.242016e−03
54	2.767465e−04	5.159447e−03
59	2.137092e−04	4.335889e−03
64	1.645170e−04	3.694694e−03
69	1.251687e−04	3.185834e−03
74	9.298583e−05	2.775359e−03
79	6.629552e−05	2.439399e−03
84	4.417904e−05	2.160923e−03
89	2.619312e−05	1.927529e−03
94	1.208936e−05	1.730009e−03
99	1.664658e−06	1.561363e−03

La prima cosa importante che si intravede è che le spline cubiche non presentano il problema del mal condizionamento in casi di una crescita del numero n delle ascisse di interpolazione. Questo perché sono definite su polinomi interpolanti a tratti di grado fisso tre. Di conseguenza Λ_n la costante di Lebesgue che rappresenta il numero di condizionamento del problema non cresce più rispetto al numero delle ascisse. Infatti le spline cubiche sono sempre ben condizionate indipendentemente dal numero delle ascisse interpolanti.

Dai risultati ottenuti si può affermare che utilizzando le spline naturali la scelta delle ascisse equidistanti è la scelta migliore poiché si commette un errore massimo più piccolo in confronto alle ascisse di Chebyshev inoltre più cresce il numero delle ascisse interpolanti, più diminuisce l'errore commesso nel interpolazione. Considerando il fatto che le spline cubiche sono sempre ben condizionate si può aumentare il numero delle ascisse di interpolazione in modo tale da diminuire l'errore commesso. Con la crescita del numero delle ascisse si ha anche un numero maggiore di calcoli (valutazioni, iterazioni) da eseguire. Questo risultato dimostra che con il crescere delle ascisse di interpolazione le spline cubiche sono la scelta migliore a confronto ai metodi di Newton e Hermite.

1.19 Esercizio 19

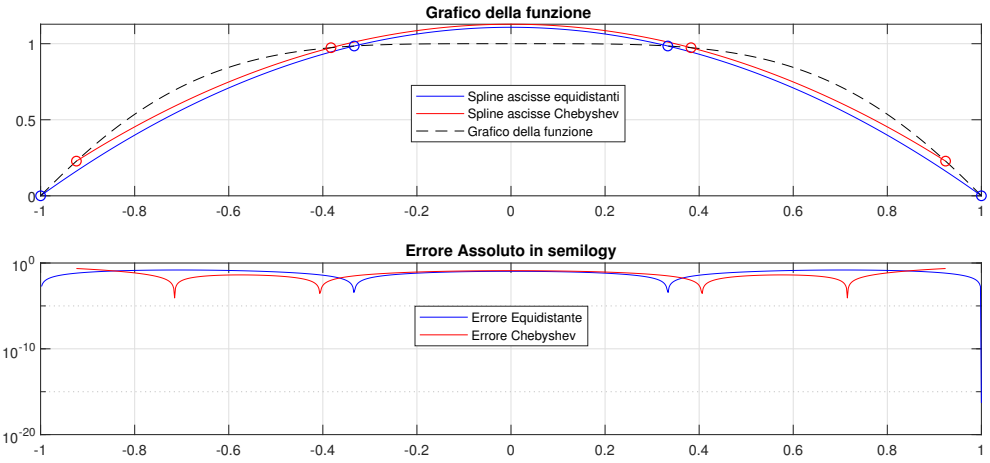
Utilizzare la function **spline** di Matlab per approssimare la funzione $f(x) = cos(\pi x^2/2)$ rispettivamente costruiti con $n+1$ ascisse equidistanti e con $n+1$ ascisse di Chebyshev sull'intervallo $[-1;1]$. Graficare (in formato **semilogy**) il massimo errore di interpolazione, per $n = 4, 5, ..., 100$. Compararli con quelli dell'esercizio precedente.

Soluzione

La funzione **spline** di Matlab implementa la spline cubica con le condizioni not-a-knot. Per costruire le ascisse di Chebyshev si usa la funzione *ceby* come precedentemente implementate inoltre i punti di valutazione vengono ridefiniti in modo tale da avere una interpolazione dei punti da valutare rispetto a le ascisse di interpolazione. Per graficare le differenze della scelta delle ascisse rispettivamente ascisse equidistanti e ascisse di Chebyshev si fa uso della funzione $e19$ indicando con $n+1$ il numero delle ascisse interpolanti.

```
function e19(n)
%   e19(n)  Funzione per calcolare i risultati del esercizio 19
%   Mostra e confronta i grafici delle funzioni costruite con la funzione spline
%   commando di Matlab, ripetivamente con le asscisse equidistanti e le ascisse di Chebyshev.
        f=@(x) cos((pi*x.^2)/2);
        a=-1;
        b=1;
        xi=linspace(a,b,n+1);
        fi=f(xi);
        k=1001;
        x=linspace(a,b,k);
        fx=f(x);
        y=spline(xi,fi,x);
        xi2=ceby(n,a,b)';
        fi2=f(xi2);
        x2=linspace(xi2(1),xi2(end),k);
        y2=spline(xi2,fi2,x2);
        figure
        subplot(2 ,1 ,1);
        plot(x,y,'b',x2,y2,'r',x,fx,'k--',xi,fi,'bo',xi2,fi2,'ro');
        title ('Grafico della funzione ');
        grid on;
        legend('Spline ascisse equidistanti','Spline ascisse Chebyshev','Grafico della funzione');
        e=abs(y-fx);
        e1=abs(y2-fx);
        subplot(2,1,2);
        semilogy(x,e,'b',x2,e1,'r');
        title ('Errore Assoluto in semilogy')
        grid on
        legend('Errore Equidistante','Errore Chebyshev');
        norm(e)
        norm(e1)
end
```

La funzione costruisce due insiemi di grafi dove nel primo insieme mette in confronto i grafi costruiti secondo le ascisse scelte tramite la funzione spline di Matlab e nel secondo insieme grafica le norme degli errori assoluti tramite il comando *semilogy*. Nella seguente figura si mettono in confronto i grafi con 4 ascisse le quali sono anche presentate sul grafo.



Invece se si desidera confrontare visivamente la norma dei errori assoluti per tutti i gradi del polinomio interpolante con ascisse differenti si fa uso della funzione *stampa*19.

```
function stampa19()
%   stampa19()  Stampa i risultati per l'esercizio 19
%Stampa in linea di commando la norma del errore assoluto della
%funzione Matlab spline rispetivamente  con ascisse equidistanti
%e  ascisse di Chebyshev
        f=@(x) cos((pi*x.^2)/2);
        a=-1;
        b=1;
        k=101;  %punti da valutare meglio dispari
```

```
        x=linspace(a,b,k);
        fprintf('\nN    ||Norma errore equidistanti|| Norma errore Spline ascisse Chebyshev');
        for n=4 : 100
                xi=linspace(a,b,n+1);
                fi=f(xi);
                fx=f(x);
                y=spline(xi,fi,x);
                xi2=ceby(n,a,b);
                fi2=f(xi2);
                x2=linspace(xi2(1),xi2(end),k);
                y2=spline(xi2,fi2,x2);
                e=norm(abs(y-fx));
                e2=norm(abs(y2-fx));
                fprintf('\n%d    ||          %d      ||              %d' ,n, e, e2);
        end
end
```

che per ovvie questione di spazio presentiamo solo una parte dei risultati.

N	Norma errore equidistanti	Norma errore Spline ascisse Chebyshev
4	4.983555e−02	6.064700e−01
9	2.165364e−02	1.538009e−01
14	4.317853e−03	6.888206e−02
19	1.248224e−03	3.887555e−02
24	4.648293e−04	2.491860e−02
29	2.052466e−04	1.731896e−02
34	1.021458e−04	1.273051e−02
39	5.544886e−05	9.749883e−03
44	3.213210e−05	7.705245e−03
49	1.960613e−05	6.242283e−03
54	1.239483e−05	5.159455e−03
59	8.030780e−06	4.335815e−03
64	5.353466e−06	3.694646e−03
69	3.666252e−06	3.185831e−03
74	2.455711e−06	2.775371e−03
79	1.583895e−06	2.439407e−03
84	9.792158e−07	2.160924e−03
89	5.728841e−07	1.927527e−03
94	3.204645e−07	1.730008e−03
99	8.277336e−08	1.561363e−03

Le spline cubiche con le condizioni not-a-knot, in caso di una crescente numero n delle ascisse interpolanti, non presenta un problema mal condizionamento . Questo perché sono definite su polinomi interpolanti a tratti di grado fisso tre. Di conseguenza Λ_n la costante di Lebesgue che rappresenta il numero di condizionamento del problema non cresce più rispetto al numero delle ascisse. Infatti le spline cubiche sono sempre ben condizionate indipendentemente dal numero delle ascisse interpolanti.

Dai risultati ottenuti si può affermare che utilizzando le spline naturali e not-a-knot la scelta delle ascisse equidistanti è la scelta migliore poiché si commette un errore massimo più piccolo in confronto alle ascisse di Chebyshev inoltre più cresce il numero delle ascisse interpolanti, più diminuisce l'errore commesso nel interpolazione. Considerando anche il fatto che le spline cubiche sono sempre ben condizionate si può aumentare il numero delle ascisse di interpolazione in modo tale da diminuire l'errore commesso. Questo risultato dimostra che con il crescere delle ascisse di interpolazione le spline cubiche sono una scelta migliore in confronto ai metodi di Newton e Hermite.
Se confrontiamo la norma dell'errore della spline naturale e not-a-knot si scopre una miglior approssimazione per la spline not-a-knot, infatti si ha una norma dell'errore minore per quest'ultima. Allora si può constatare che le condizioni della spline not-a-knot influenzano meglio l'approssimazione tramite spline cubica in confronto a le condizioni della spline naturale.

1.20 Esercizio 20

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

$$\tilde{f}(x) = f(x) + 10^{-3} rand(size(x)),$$

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

$$x_i = -1 + 2i/n, \quad n = 10^4$$

Graficare (in formato semilogy) l'errore di approssimazione $||f - p||$ relativo all'intervallo $[-1; 1]$, rispetto ad m, per $m = 1, 2...20$. Commentare i risultati ottenuti.

Soluzione

Per l'approssimazione polinomiale ai minimi quadrati si usa la seguente *function* **e20**. La funzione prende come input il grado del polinomio interpolante.

```
function [] = e20(m)
%   e20(m)
%Funzione per la soluzione del esercizio 20. Esegue l'approssimazione della
%funzione data tramite il metodo dei minimi quadrati. Confronta il grafico
%della approssimazione e del errore rispetivo.
% Input:
% - m: Grado desiderato della funzione da approssimare
f=@(x) cos((pi*x.^(2))./2);
n=10000; %numero delle ascisse da valutare
x=zeros(1,n)';
for i = 0 : n-1
        x(i+1)= -1 + (2*(i))/(n-1);
end
fx=feval(f,x);
f1=fx+10^(-3).*rand(size(x));
V=fliplr(vander(x));%Costruzione della matrice Vandermonde
V=V(:,1:m+1);
V=myqr(V);
p=qrsolve(V,f1); %Soluzione tramite fattorizzazione qr
y=valuta(flip(p),x);
subplot(2 ,1 ,1);
plot(x,f1,'b',x,y,'r');
title ('Approssimazione minimo quadrati');
legend('Grafo funzione perturbata','Grafico della funzione approssimata');
e=abs(fx-f1);
e1=abs(fx-y);
subplot(2 ,1 ,2);
semilogy(x,e,'b',x,e1,'r');
title ('Errore di approssimazione in semilogy')
legend('Errore funzione perturbata','Errore funzione approssimata');
norm(e)
norm(e1)
end
```

Il metodo dei minimi quadrati consiste nel trovare il vettore a che minimizza la quantità

$$||y - z||_2^2$$

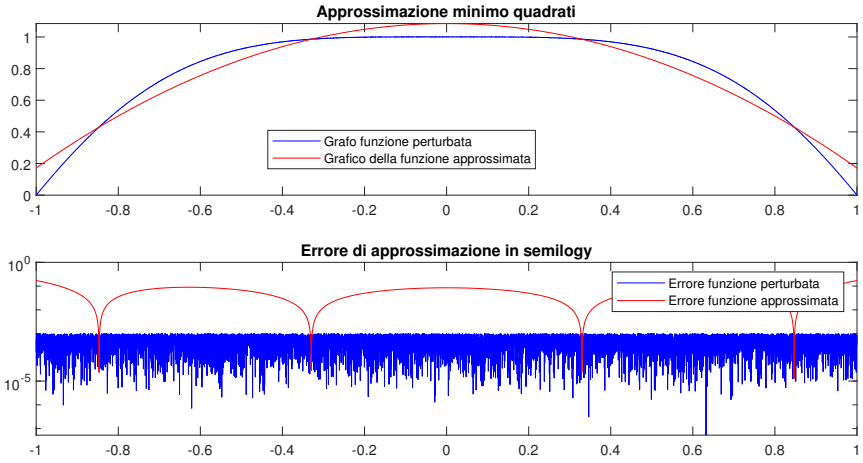
Il problema della determinazione del polinomio interpolante è equivalente a risolvere nel senso dei minimi quadrati il sistema sovradeterminato

$$Va = y$$

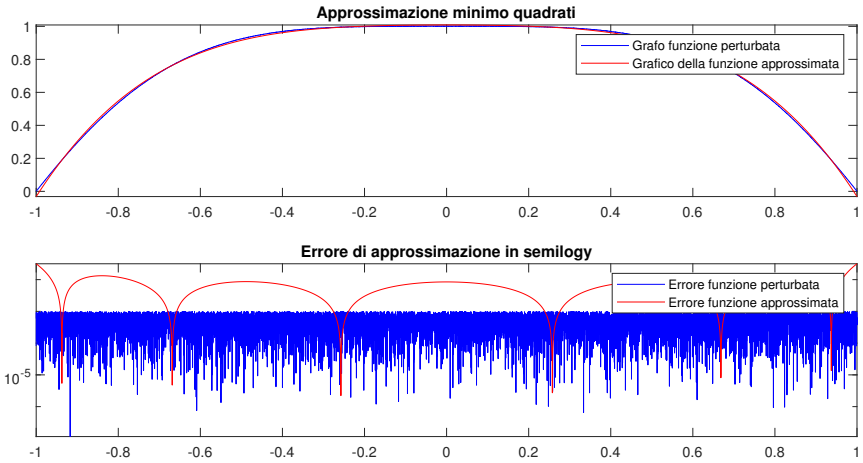
La *function* costruisce il vettore delle ascisse x da valutare inoltre costruisce la matrice di Vandermonde V in base al grado desiderato. Una delle proprietà delle matrici di Vandermonde è che hanno rango massimo di conseguenza si può usare la fattorizzazione QR per la soluzione del sistema sovradeterminato nel senso dei minimi quadrati. Per la soluzione dei sistema sovradetermino si usano le *function* del esercizio 11 e 12. Per la valutazione delle ascisse in base ai coefficienti polinomiali trovati si fa uso della *function* **valuta** che restituisce i punti della funzione approssimata.

```
function [y] = valuta(p,x)
%   valuta(p,x)
% % Valutazione delle ascisse x tramite i coefficenti del polinomio in modo
% tale che y=p(x).
% Input:
% - p: coefficienti polinomiali
% - x: ascisse della funzione da valutare.
% Output:
% - y : coordinate dell polinomio valutati.
a = length(p);
y = zeros(size(x));
if a > 0
y(:) = p(1);
end
for i = 2:a
        y = x .* y + p(i);
end
end
```

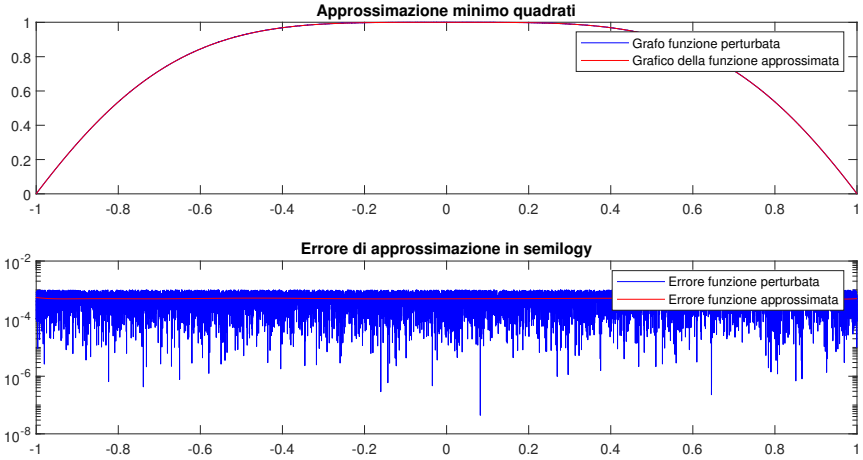
La *function* **e20** inoltre costruisce due insiemi di grafi dove nel primo insieme mette a confronto i grafici costruiti secondo dati perturbati e i risultati del metodo, e nel secondo insieme grafica gli errori assoluti tramite il comando *semilogy*.
In seguito si confronta i grafi con un crescente grado della approssimazione. Nella seguente figura si confrontano i grafi in qui si una un grado $m = 2$ di approssimazione.



Come si vede non si ha una buona approssimazione. Vediamo i risultati con un grado $m = 5$.



In questo caso si ha una buona approssimazione anche se confrontando l'errore si vede un miglioramento del confronto precedente ma ancora lontano da dire buono. Prendendo un grado $m = 10$ si ha il seguente risultato



Si ha una migliore approssimazione e inoltre l'errore commesso è minore dell'errore della funzione perturbata. Si può affermare che la funzione approssimata, approssima meglio della funzione perturbata.

1.21 Esercizio 21

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 (come numeri razionali).

Soluzione

Da le formule di Newton-Cotes si ha la seguente formula dei pesi:

$$c_{kn} = \int_0^n \prod_{j=0, j\neq k}^{n} \frac{t-j}{k-j} dx$$

La *function* **pesi** prende come input il grado della funzione di Newton-Cotes, $n \in N, n > 0$, e calcola i pesi rispettivi, restituendo un vettore di numeri razionali.

```
function P = pesi(n)
%    pesi(n) - Funzione che calcola i pesi di Newton - Cotes. Dato in ingresso
% il grado N la funzione calcola i pesi della formula rispetiva. Serve per
% la soluzione del esercizio 21.
% Input:
% - n: Grado della funzione esempio (n= 1 ....50)
% Output:
% - P: il vettore di numeri razionali dei pesi della NC
        m=n+1;
        P=sym('t' , [m 1]);
        syms t; % Per avere numeri razionali
        if mod(m,2)==0 %Se il numero dei pesi e pari
                for k=0 : m/2-1
                        i=k+1;
                        P(i)=1;
                        j=0;
                        while j <= m-1
                                if j==k
                                        j=j+1;
                                else
                                        P(i)=(P(i)*(t-j))/(k-j);
                                j=j+1;
```

```
                        end
                end
        end
        a=1;
        for k=m/2+1 : m %P(k)= P(k-a)
                P(k)=P(k-a);
                a=a+2;
        end
        else    %Se il numero dei pesi e dispari
                for k=0 : m/2
                        i=k+1;
                        P(i)=1;
                        j=0;
                        while j <= m-1
                                if j==k
                                        j=j+1;
                                else
                                        P(i)=(P(i)*(t-j))/(k-j);
                                j=j+1;
                                end
                        end
                end
                a=2;
                for k=n/2+2 : m
                        P(k)=P(k-a);
                        a=a+2;
                end
        end
        for k=1 : m
        P(k)=int(P(k),0,n); % calcoliamo l integrale per definire i pesi
        end
end
```

La *function* può essere implementata più efficacemente se non si considerano i valori simbolici inoltre considerando che i pesi vendono calcolati una volta sola si possono memorizzare a parte senza ulteriormente chiamare la *function*. Per tabulare i pesi per $n = 1.2...7$ si fa uso della *function* **e21()** che stampa i vettori di numeri razionali rispettivi.

```
function e21()
% e21() - Funzione per la risoluzione del esercizio 21. Stampa nella linea
%di commando i pesi delle formule di grado 1.2...7 in numeri razionali-
        fprintf('\nN - Pesi delle formule di Newton Cotes come numeri razionali\n');
        for n=1 : 7
        fprintf('%d - ',n);
        disp(pesi(n)'); %La funzione che calcola i pesi di NC
        end
end
```

La funzione stampa il la seguente schermata:

```
N - Pesi delle formule di Newton Cotes come numeri razionali
1 - [ 1/2, 1/2]

2 - [ 1/3, 4/3, 1/3]

3 - [ 3/8, 9/8, 9/8, 3/8]

4 - [ 14/45, 64/45, 8/15, 64/45, 14/45]

5 - [ 95/288, 125/96, 125/144, 125/144, 125/96, 95/288]

6 - [ 41/140, 54/35, 27/140, 68/35, 27/140, 54/35, 41/140]

7 - [ 5257/17280, 25039/17280, 343/640, 20923/17280, 20923/17280, 343/640, ...
      25039/17280, 5257/17280]
```

1.22 Esercizio 22

Utilizzare la function del precedente esercizio per graficare, in formato semilogy, il rapporto $\kappa_n = \kappa$ rispetto ad n, essendo κ il numero di condizionamento di un integrale definito, e κ_n quello della formula di Newton-Cotes utilizzata di grado n per approssimarlo. Riportare i risultati per n = 1.... 50.

Soluzione

Dal teorema 5.1 si ha:

$$\frac{1}{n}\sum_{k=0}^{n} c_{kn} = 1$$

Considerando il rapporto fra condizionamento del problema di Newton-Cotes e il condizionamento generale del problema del calcolo del integrale si ha

$$\frac{k_n}{k} = \frac{\frac{(b-a)}{n}\sum_{k=0}^{n}|c_{kn}|}{(b-a)} = \frac{\sum_{k=0}^{n}|c_{kn}|}{n}$$

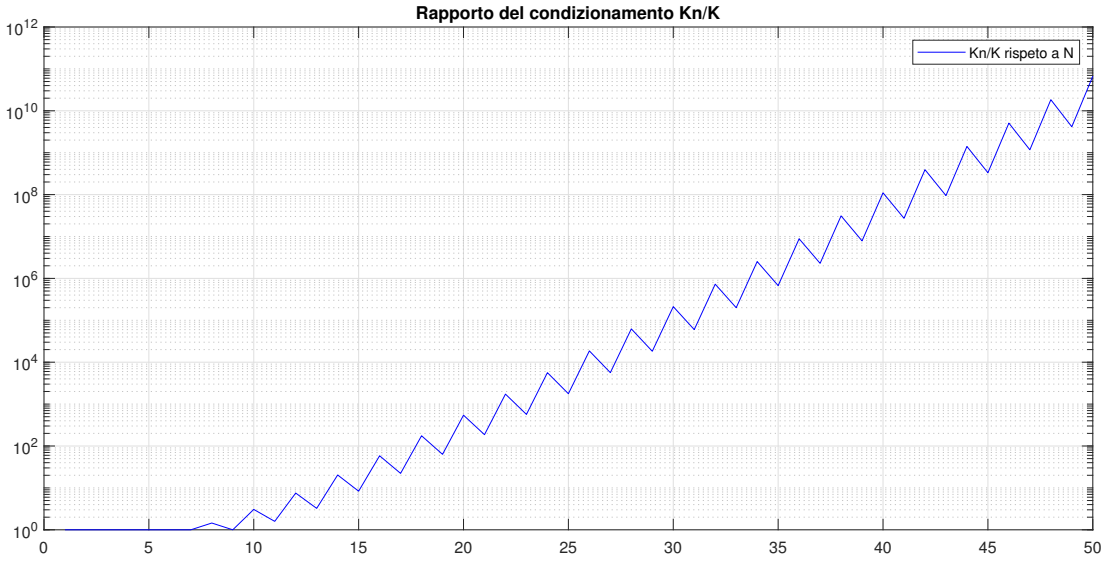
dove distinguiamo due casi

- per ogni $i = 0, 1, ...n$ se $0 \leq c_{in}$ allora $\frac{1}{n}\sum_{k=0}^{n}|c_{kn}| = \frac{1}{n}\sum_{k=0}^{n} c_{kn} = 1$ da qui per pesi positivi si ha $\kappa_n = \kappa$
- se esiste i tale $c_{in} \leq 0$ allora $\frac{1}{n}\sum_{k=0}^{n}|c_{kn}| > 1$ pertanto se esiste anche un peso negativo $\kappa_n > \kappa$

Per costruire il grafo richiesto si fa uso del *fanction* **e22**

```
function  e22(m)
% e22() - Funzione per la risoluzione del esercizio 22. Mostra il grafo del
% rapporto fra kn/k per i = 1 .... m . Per l esercizio e richiesto m=50.
% Input:
% - m: Naturale fino a che grado confrontare i condizionamenti
        kn=zeros(1,m);
        for n =1 : m
                tmp=abs(pesi(n));
                for j=1 : length(tmp)
                kn(n)=kn(n)+tmp(j);
        end
        kn(n)=kn(n)/n;
        end
        semilogy([1:m],kn,'b');
        title ('Rapporto del condizionamento Kn/K');
        grid on;
        legend('Kn/K rispeto a N');
end
```

che mostra il seguente risultato



Come ci si aspetta per $n = 1, 2..7$ e 9 si ha che $\kappa_n = \kappa$ invece per altri valori di n il rapporto dimostra un crescente valore di k_n. Concludendo, le formule di Newton Cotes sono convenienti per i valori di $n = 1, 2..7$ e 9, invece per gli altri valori si è in presenza di mal condizionamento del problema di conseguenza non sono consigliati.

1.23 Esercizio 23

Tabulare le approssimazioni dell'integrale

$$I(f) = \int_{-1}^{1.1} tan(x)dx \equiv log\frac{cos(1)}{cos(1.1)}$$

ottenute mediate le formule di Newton-Cotes di grado n, $n = 1....9$. Tabulare anche il relativo errore (in notazione scientifica con 3 cifre significative).

Soluzione

Per il calcolo dell'integrale secondo le formule di Newton Cotes usando i pesi definiti nel esercizio precedente si fa uso della seguente *function* **nc**.

```
function I=nc(f,a,b,n)
% nc(f,a,b,n) - Calcola l integrale della funzione f nel intervallo a , b
% usando le formule di Newton Cotes di grado n
% Input:
% - f: funzione da integrare
% - a: estemo inferiore della funzione integrana
% - b: estemo superiore della funzione integrana
% - n: Grado delle formule di NC
% Output:
% - I: Integrale della funzione
        if n>=1 & n<=40 %Per la relazione e richesta calcolo fino a 9
                w=pesi(n)/n;
                x=linspace(a,b,n+1)';
                vf=feval(f,x);
                I=(b-a)*(eval(w)'*vf);
        else
                disp('Si deve scegliere grado di valore fra 1 e 9 compresi');
end
```

Per la soluzione dell'esercizio si usa la *function* **23**

```
function e23()
% e23() - Funzione per la risoluzione del esercizio 23. Stampa in console il
% calcolo del integrale e lerrore relativo in base al grado n =1...9
        f=@(x) tan(x);
        If=log(cos(1)/cos(1.1));
        I=ones(0,9);
        E=ones(0,9);
        for i=1 : 9
                I(i)=nc(f,-1,1.1,i);
                E(i)=If-I(i);
                end
                fprintf('\nN || Approssimazione Integrale ||Errore di approssimazione\n');
                for i=1 : 9
                fprintf('%d ||                %.3f        ||       %.3f \n',i,I(i),E(i));
        end
end
```

che stampa su console il seguente risultato.

N	Approssimazione Integrale	Errore di approssimazione
1	0.428	−0.253
2	0.213	−0.038
3	0.196	−0.021
4	0.180	−0.006
5	0.179	−0.004
6	0.176	−0.001
7	0.176	−0.001
8	0.175	−0.000
9	0.175	−0.000

Per valori crescenti di n si ha una migliore approssimazione dell'integrale. Infatti l'errore diminuisce per valori crescenti di n

1.24 Esercizio 24

Confrontare le formula composite dei trapezi e di Simpson per approssimare l'integrale del precedente esercizio, per valori crescenti del numero dei sottointervalli dell'intervallo di integrazione. Commentare i risultati ottenuti, in termini di costo computazionale.

Soluzione

Per tabulare i risultati del confronto della formula composita dei trapezi e di Simpson si fa uso della seguente *function* **e24**

```
function e24(n)
%       e24()
% Stampa su linea di commando i risultati richiesti dal esercizio 24.
% Input:
% - n: Numero dei sottointervalli del intervallo di integrazione
f=@(x) tan(x);
ValoreFunzione=log(cos(1)/cos(1.1));

fprintf('\nN  ||  Integrale I1n        || Integrale I2n\n');
for i=1 : n
        I1n=trapecomp(f,-1,1.1,i);
        if mod (i, 2) == 0
                I2n=simpcomp(f,-1,1.1,i);
                fprintf('%d  ||        %f        ||       %f \n',i,I1n,I2n);
        else
                fprintf('%d  ||        %f        ||       ---- \n',i,I1n);
        end
end
ValoreFunzione
```

La *function* prende come input un un numero $n \in N$ e stampa il risultato dell'integrazione della funzione per la formula composita dei trapezi e di Simpson con sottointervalli $k = 1, 2...n$. In seguito un esempio per un sottointervallo fino a $n = 10$

```
N    ||   Integrale I1n        || Integrale I2n
1    ||        0.427720        ||      ------
2    ||        0.266404        ||       0.212632
3    ||        0.221993        ||      ------
4    ||        0.203433        ||       0.182443
5    ||        0.193961        ||      ------
6    ||        0.188498        ||       0.177333
7    ||        0.185073        ||      ------
8    ||        0.182789        ||       0.175908
9    ||        0.181193        ||      ------
10   ||        0.180035        ||       0.175393

ValoreFunzione =
0.174921606868218
```

Le formule composite hanno un costo lineare. Sapendo che le somme sono utilizzano le operazione vettoriali si può dire che esse sono insignificanti in termini computazionali. Questo risultato vale sia per la formula composita dei trapezi, sia per la formula composita di Simpson.
Le operazioni più onerose sono quelle della valutazione della funzione anche esse sono lineari al numero dei sottointervalli, precisamente $n + 1$. Nel nostro esempio, visto che si tratta della stessa funzione si può dire che in termini computazionali le due formule sono equivalenti.
Dal confronto dei risultati si nota che la formula composita di Simpson ha una approssimazione migliore. Inoltre tende più velocemente al risultato in confronto alla formula composita dei trapezi. Concludendo, la formula composita di Simpson è migliore della formula composita dei trapezi.

Per il calcolo dell'integrale della formula dei trapezi si utilizza la formula vista in lezione e qui in seguito

```
function [ I ] = trapecomp( fun, a, b, n)
% [ I ] = trapecomp( f, a, b, n)
% Ccalcolo dell'integrale di una funzione in un dato
% intervallo [a, b] utilizzando la formula dei trapezi composita.
% Input:
% - f: stringa con il nome della funzione che implementa la funzione
%       integranda
% - a: estremo sinistro dell'intervallo
% - b: estremo destro dell'intervallo
% - n: numero desiderato di partizioni nell'intervallo [a, b]
% Output:
% -I: l'area approssimata
        if a==b
                I=0;
        elseif n<1 || n ~= fix(n)
                errore('Numero di ascisse non valide');
        else
                h=(b-a)/n;
                x=linspace(a,b,n+1);
                f=feval(fun,x);
                I=h*(f(1)/2+sum(f(2:n))+f(n+1)/2);
        end
        return
end
```

Per il calcolo dell'integrale della formula di Simpson si utilizza la formula vista in lezione e qui in seguito

```
function [I] = simpcomp(fun,a,b,n)
% [ I ] = simpcomp( f, a, b, n)
%     simpcomp(@sin,0,1,6) Esempio
% Algoritmo per il calcolo dell'integrale di una funzione in un dato
% intervallo [a, b] utilizzando la formula dei trapezi composita.
% Input:
% - f: stringa con il nome della funzione che implementa la funzione
```

```
%       integranda
% - a: estremo sinistro dell'intervallo
% - b: estremo destro dell'intervallo
% - n: numero desiderato di partizioni nell'intervallo [a, b] (numero pari)
% Output:
% -I: l'area approssimata
        if a==b
                I=0;
        elseif n<2 || n/2 ~= fix(n/2)
                errore('Numero di ascisse non valide');
        else
                h=(b-a)/n;
                x=linspace(a,b,n+1);
                f=feval(fun,x);
                I=(h/3)*(f(1)+f(n+1)+4*sum(f(2:2:n))+2*sum(f(3:2:n-1)));
        end
        return
end
```

1.25 Esercizio 25

Confrontare le formule adattive dei trapezi e di Simpson, con tolleranze $tol \quad 10^2, 10^3 ....10^6$, per approssimare l'integrale definito

$$I(f) = \int_{-1}^{1} \frac{1}{1+10^2 x^2} dx$$

Commentare i risultati ottenuti, in termini di costo computazionale.

Soluzione

Per l'implementazione della formula adattiva dei trapezi si utilizza la seguente *function* **adaptrap**

```
function [I2,points] = adaptrap( f, a, b, tol, fa, fb )
% [I2,points] = adaptrap( f, a, b, tol )
% Calcolo dell'integrale di una funzione in un dato
% intervallo [a, b] utilizzando la formula adattiva dei trapezi.
% Input:
% - f: stringa con il nome della funzione che implementa la funzione
%       integranda
% - a: estremo sinistro dell'intervallo
% - b: estremo destro dell'intervallo
% - tol: toleranza richiesta per il calcolo
% - fa: Valore della funzione  per la chiamata ricorsiva
% - fb: Valore della funzione  per la chiamata ricorsiva
% Output:
% - I2: l'area approssimat
% - points: L'insieme delle ascisse e delle coordinate dei punti valutati
        global points   %opzionale
        delta = 0.5; % ampiezza minima intervalli
        if nargin<=4   %numero argumenti 4 chiamata function del utente
                fa = feval( f, a );
                fb = feval( f, b );
                if nargout==2
                        points = [a fa; b fb];
                else
                points = [];
                end
        end
        h = b-a;
        x1 = (a+b)/2;
        f1 = feval( f, x1 );
        if ~isempty(points)
        points = [points; [x1 f1]];  %contateniamo nuovo valore
        end
        I1 = .5*h*( fa+fb );
        I2 = .5*( I1 + h*f1 );
```

```
        e = abs( I2-I1 )/3;
        if e>tol || abs(b-a)>delta
                I2 = adaptrap( f, a, x1, tol/2, fa, f1 ) +...
                adaptrap( f, x1, b, tol/2, f1, fb );
        end
        return
end
```

invece per l'implementazione della formula adattiva di Simpson si fa uso della *function* **adapsim**

```
function [I2,points] = adapsim( f, a, b, tol, fa, f1, fb )
% [I2,points] = adapsim( f, a, b, tol )
% Calcolo dell'integrale di una funzione in un dato
% intervallo [a, b] utilizzando la formula adattiva dei trapezi.
% Input:
% - f: stringa con il nome della funzione che implementa la funzione
%       integranda
% - a: estremo sinistro dell'intervallo
% - b: estremo destro dell'intervallo
% - tol: toleranza richiesta per il calcolo
% - fa: Valore della funzione  per la chiamata ricorsiva
% - f1: Valore della funzione  per la chiamata ricorsiva
% - fb: Valore della funzione  per la chiamata ricorsiva
% Output:
% - I2: l'area approssimat
% - points: L'insieme delle ascisse e delle coordinate dei punti valutati
        global points
        delta = 0.5; % ampiezza minima intervalli
        x1 = (a+b)/2;
        if nargin<=4
                fa = feval( f, a );
                fb = feval( f, b );
                f1 = feval( f, x1 );
                if nargout==2
                        points = [a fa;x1 f1; b fb];
                else
                        points = [];
        end
        end
        h = (b-a)/6;
        x2 = (a+x1)/2;
        x3 = (x1+b)/2;
        f2 = feval( f, x2 );
        f3 = feval( f, x3 );
        if ~isempty(points)
        points = [points; [x2 f2; x3 f3]];
        end
        I1 = h*( fa+4*f1+fb );
        I2 = .5*h*( fa + 4*f2 + 2*f1 + 4*f3 +fb );
        e = abs( I2-I1 )/15;
        if e>tol || abs(b-a)>delta
                I2 = adapsim( f, a, x1, tol/2, fa, f2, f1 ) + ...
                adapsim( f, x1, b, tol/2, f1, f3, fb );
        end
        return
end
```

I metodi dal punto di vista computazionale a parità di nodi sono quasi identici con una leggera efficienza per la formula adattiva dei trapezi. Considerando il fatto che i metodi sono costruiti in modo tale che le operazioni di somma e moltiplicazione siano operazioni vettoriali si ha come risultato che queste operazioni non influenzano molto su le prestazioni del calcolo. Le operazioni più onerose sono quelle di valutazione della funzione per questo il metodo più efficiente sarà il metodo che genera meno nodi. In seguito si confrontano i metodi con valori differenti di tolleranza.

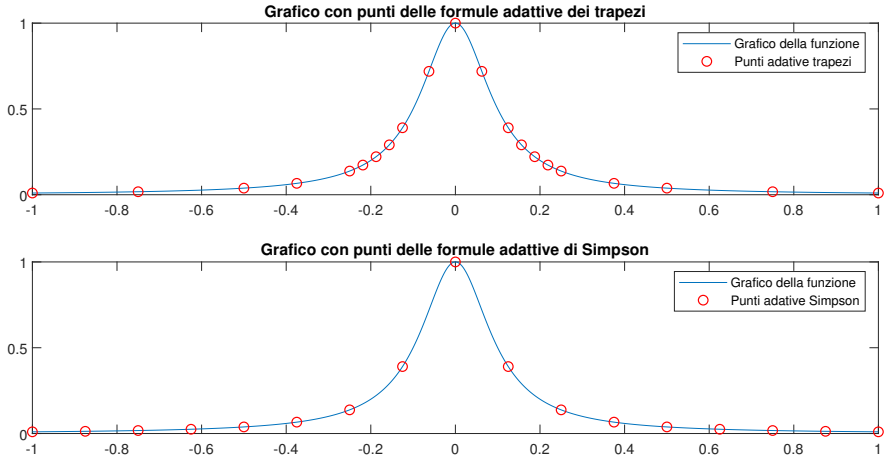
Per il confronto dei due metodi si fa uso della *function* **e25** che prende come input una tolleranza $tol = 10^{-i}, i = 2, 3, 4, 5, 6$

```
function e25(tol)
%   e25(tol)
%La funzione che implementa la risoluzione del esercizio 25.
%Prende come input una tolleranza e mette in confronto le formule adattive
%dei trapezi e di Simpson stampando in linea di comando il numero dei punti
```

```
%e graficando i rispetivi punti sul grafico della funzione del esercizio.
% Input:
% - tol: Toleranza richiesta per il confronto (Es: 10.^(-2)
        f=@(x) 1./(1+(10.^2).*(x.^2));
        x=linspace(-1,1,1001);
        y=feval(f,x);
        fprintf('\n   tol    ||Adattive Trapezi|Numero punti||Adattive Simpson |Numero punti\n');
        [I1,P1]=adaptrap(f,-1,1,tol);
        [I2,P2]=adapsim(f,-1,1,tol);
        fprintf('%f ||   %f     |   %d        ||    %f     |   %d\n',tol,I1,length(P1),I2,length(P2
            ));
        figure
        subplot(2 ,1 ,1);
        plot(x,y,P1(1:length(P1),1),P1(1:length(P1),2),'ro');
        title ('Grafico con punti delle formule adattive dei trapezi');
        legend('Grafico della funzione','Punti adative trapezi');

        subplot(2 ,1 ,2);
        plot(x,y,P2(1:length(P2),1),P2(1:length(P2),2),'ro');
        title ('Grafico con punti delle formule adattive di Simpson');
        legend('Grafico della funzione','Punti adative Simpson');
end
```

La *function* confronta i metodi calcolando l'integrale e mostrando graficamente i punti della funzione valutati. Inoltre stampa su console il numero dei nodi necessari per ogni metodo.
Per una $tol = 10^{-2}$ si ha il seguente risultato

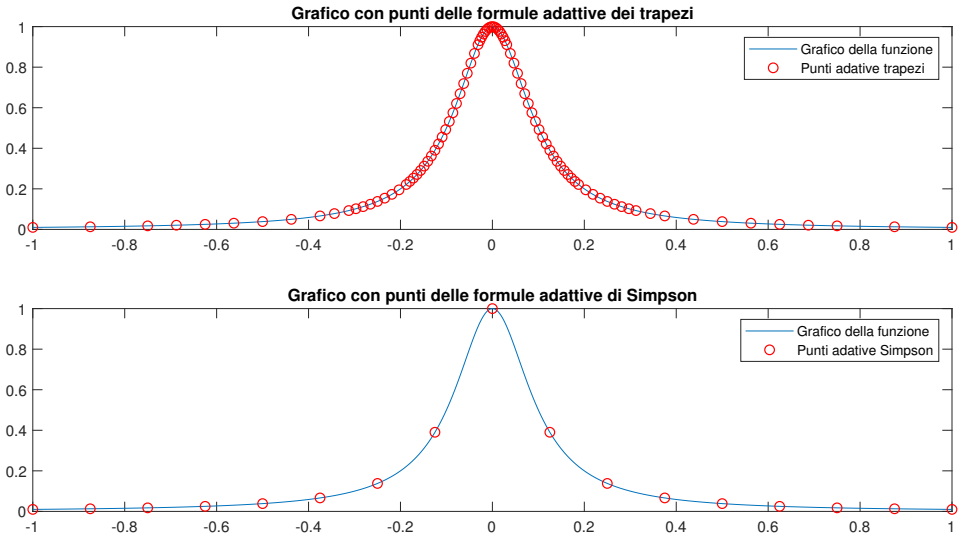


e stampa su console

```
    tol     ||Adattive Trapezi|Numero punti||Adattive Simpson |Numero punti
0.010000 ||    0.295560      |    21        ||     0.281298      |    17
```

Con una tolleranza di questo ordine i numero dei nodi per i metodi è quasi identico ma considerando che l'integrale della funzione vale 0.2942255 si vede una migliore approssimazione per la formula adattiva dei trapezi.

Per una $tol = 10^{-3}$ si ha il seguente risultato

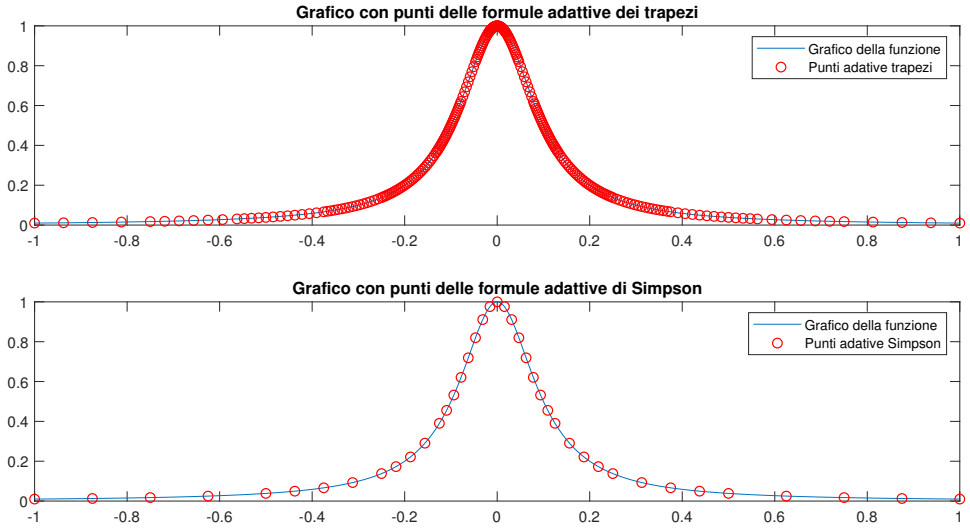


e stampa su console

tol	Adattive Trapezi	Numero punti	Adattive Simpson	Numero punti
0.001000	0.294585	93	0.281298	17

Come si vede con una tolleranza di questo ordine si ha un esplosione dei punti di conseguenza di valutazioni della funzione per il metodo adattivo dei trapezi. Invece il metodo adattivo di Simson non cambia il numero dei nodi e si può dire che si comporta meglio.

Per una $tol = 10^{-4}$ si ha il seguente risultato

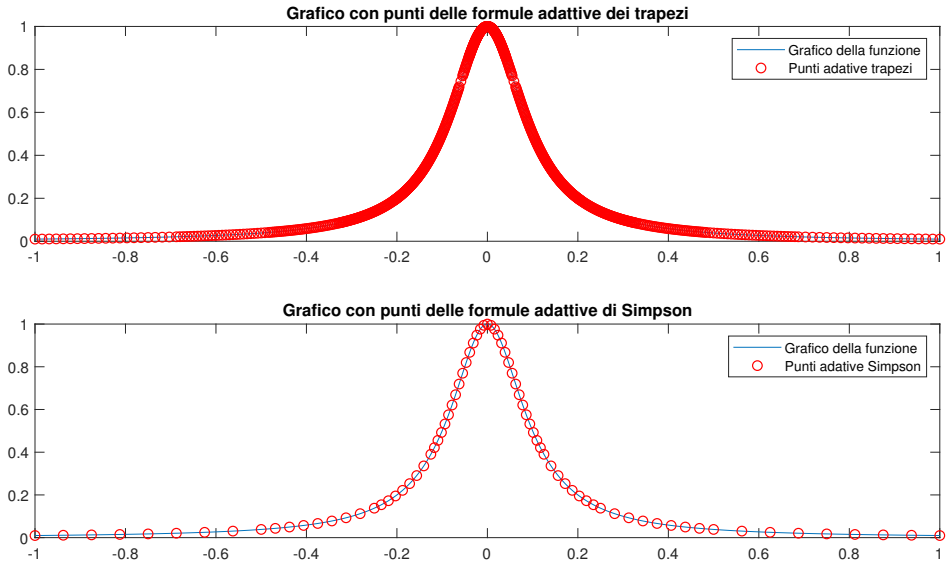


e stampa su console

tol	Adattive Trapezi	Numero punti	Adattive Simpson	Numero punti
0.000100	0.294274	277	0.294259	41

Con una tolleranza di questo ordine si ha una crescita di tre volte dei nodi rispettivamente alla tolleranza precedente ma si intravede un miglioramento nella approssimazione del metodo adattivo di Simpson.

Per una $tol = 10^{-5}$ si ha il seguente risultato

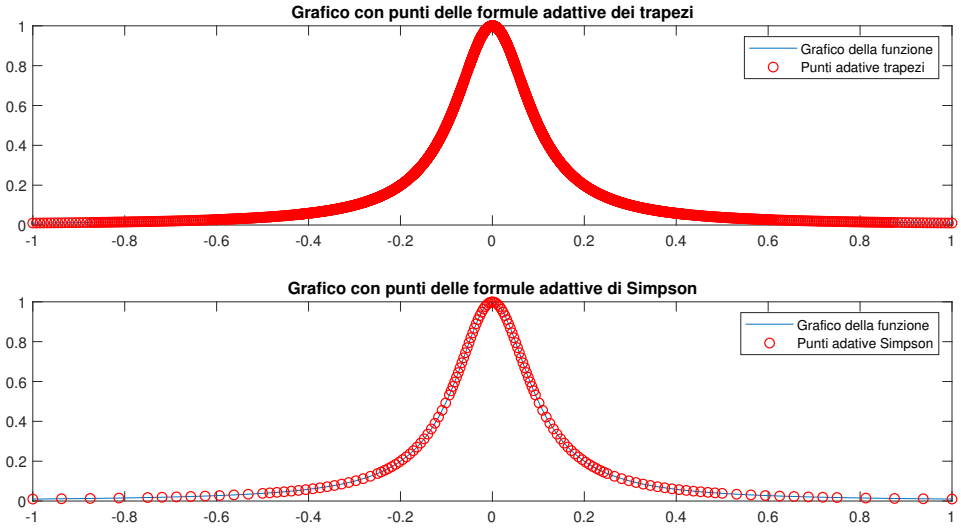


e stampa su console

tol	Adattive Trapezi	Numero punti	Adattive Simpson	Numero punti
0.000010	0.294230	793	0.294228	81

Con questo ordine di tolleranza si ha un esplosione del numero dei nodi per la formula adattiva dei trapezi invece la formula adattiva di Simpson ha un numero sostenuto di nodi è inoltre ha una approssimazione migliore.

Per una $tol = 10^{-6}$ si ha il seguente risultato



e stampa su console

tol	Adattive Trapezi	Numero punti	Adattive Simpson	Numero punti
0.000001	0.294226	2693	0.294226	145

Con una tolleranza di questo ordine diventa evidente il fatto che proseguire ad utilizzare la formula adattiva dei trapezi comporta delle computazioni onerose. Invece il metodo adattivo di Simson ha un numero di nodi

basso. Considerando il fatto che l'elemento chiave del confronto dei metodi sia il numero dei nodi generati cioè il numero delle valutazioni della funzione su questi punti si determina che per una migliore approssimazione del integrale della funzione la formula adattiva di Simson è più efficiente della formula adattiva dei trapezi.

Capitolo 2

Capitolo 2

2.1 Manuale d'uso

La seguente relazione viene accompagnata con i file sviluppati in Matlab per la risoluzione dei esercizi. I file sono organizzati nelle cartelle nel modo seguente

- Gli esercizi che non hanno bisogno del file in Matlab non hanno cartella di riferimento
- Gli esercizi che vengono accompagnati da file in Matlab hanno la cartella di riferimento. Per esempio l'esercizio 4 ha di riferimento la cartella $e4$ che contiene il file per la risoluzione.
- Se più esercizi usano gli stessi file in Matlab vengono raggruppati nella stessa cartella il cui nome include il numero dell'esercizio, per esempio gli esercizi $8, 9$ e 10 sono accompagnati dalla cartella $e11 - e12 - e13$ che contiene tutti i file per le risoluzioni.