UNIVERSITÀ
DEGLI STUDI
FIRENZE

Anno Accademico 2021/2022

Elaborato Calcolo Numerico

Autore: **Alberto Biliotti**

Matricola: **7026939**

Notare che le function riportate negli esercizi differiranno, in minima parte, da quelle riportate nella cartella allegata contenente i codici Matlab, a causa di caratteri speciali che non sono riportabili negli script Matlab di LaTex, in particolare le vocali accentate che sono state sostituite dalla stessa lettera accentata (ad esempio è diventa e')

Esercizio 1
Verificare che:

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

Soluzione
Per verificare l'equazione basta scrivere gli sviluppi di Taylor centrati in x delle funzioni presenti:

$$f(x+2h) = f(x)+2hf'(x)+2h^2f''(x)+\frac{4}{3}h^3f'''(x)+\frac{2}{3}h^4f''''(x)+O(h^5)$$

$$f(x+h) = f(x)+hf'(x)+\frac{1}{2}h^2f''(x)+\frac{1}{6}h^3f'''(x)+\frac{1}{24}h^4f''''(x)+O(h^5)$$

$$f(x-h) = f(x)-hf'(x)+\frac{1}{2}h^2f''(x)-\frac{1}{6}h^3f'''(x)+\frac{1}{24}h^4f''''(x)+O(h^5)$$

$$f(x-2h) = f(x)-2hf'(x)+2h^2f''(x)-\frac{4}{3}h^3f'''(x)-\frac{2}{3}h^4f''''(x)+O(h^5)$$

Riscrivendo il numeratore del primo membro dell'equazione sostituendo le funzioni con gli sviluppi appena ottenuti:

$$-f(x+2h)+8f(x+h)-8f(x-h)+f(x-2h) =$$

$$= -f(x)-2hf'(x)-2h^2f''(x)-\frac{4}{3}h^3f'''(x)-\frac{2}{3}h^4f''''(x)+8f(x)+8hf'(x)+4h^2f''(x)+\frac{4}{3}h^3f'''(x)+\frac{1}{3}h^4f''''(x)-8f(x)+$$

$$+8hf'(x)-4h^2f''(x)+\frac{4}{3}h^3f'''(x)-\frac{1}{3}h^4f''''(x)+f(x)-2hf'(x)+2h^2f''(x)-\frac{4}{3}h^3f'''(x)-\frac{2}{3}h^4f''''(x)+O(h^5) =$$

$$= -12hf'(x)+O(h^5)$$

Quindi il primo mebro dell'equazione diventa:

$$\frac{-f(x+2h)+8f(x+h)-8f(x-h)+f(x-2h)}{12h} = \frac{-12hf'(x)+O(h^5)}{12h} = -f'(x)+O(h^4)$$

Come volevamo dimostrare.

Esercizio 2
Calcolare, motivandone i passaggi, la precisione di macchina della doppia precisione dello standard IEEE. Confrontare questa quantità con quanto ritornato dalla variabile eps di Matlab, commentando a riguardo.

Soluzione
Sapendo che la precisione di macchina u (nel caso si usi il l'arrrotondamento) si ottiene come:

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

dove b è la base scelta mentre m è il numero di cifre usate per la mantissa. Nello standard IEEE la base è 2 mentre la mantissa è composta da 52 cifre per cui la precisione di macchina sarà:

$$u = \frac{1}{2}2^{-51} = 2^{-52}$$

che è lo stesso valore restuito dalla variabile eps di Matlab che contiene appunto la precisione di macchina del calcolatore che lavora con lo standard IEEE in doppia precisione. La precisione di macchina è definita come il massimo errore relativo dovuto alla rappresentazione in aritmetica finita di un numero reale.

Esercizio 3

Eseguire il seguente script Matlab:

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

Spiegare i risultati ottenuti.

Soluzione

Il risultato è 0.9992007221626409, tale risultato (diverso da quello atteso di 1) è dovuto al fatto che la somma algebrica sia mal condizionata nel caso di somma di numeri opposti, infatti definito k il fattore di amplificazione dell'errore (ovvero la misura di quanto gli errori iniziali possano amplificarsi nel risultato finale) si ha:

$$k = \frac{|x_1| + |x_2|}{|x_1 + x_2|}$$

Tenendo conto che il massimo errore di rappresentazione nell'aritmetica finita è la precisione di macchina u(dovuta al fatto che 10^{-14} non è un numero di macchina non essendo rappresentabile direttamente in base 2), che nel nostro caso è dell'ordine di 10^{-16} e che k per questa operazione è dell'ordine di 10^{14} ci aspetteremo un errore relativo massimo dell'ordine di 10^{-2} corrispondente a $k * u$, il che è in accordo con l'errore che abbiamo ottenuto che è dell'ordine di 10^{-4}.

Esercizio 4
Scrivere una function Matlab, radice(x) che, avendo in ingresso un numero x non negativo, ne calcoli la radice quadrata utilizzando solo operazioni algebriche elementari, con la massima precisione possibile. Confrontare con la function sqrt di Matlab per 20 valori di x, equispaziati logaritmicamente nell'intervallo [1e-10,1e10], evidenziando che si è ottenuta la massima precisione possibile.

Soluzione
La function è stata ottenuta sfruttando il metodo di Newton per cercare le radici della funzione $f(x) = x^2 - x_0$ dove x_0 è il numero di cui vogliamo conoscere la radice, infatti la la radice di tale equazione non sarà altro che radice quadrata stessa. Sapendo ciò è bastato calcolare manualmente la derivata di tale funzione che è $f'(x) = 2x$ per potermi ricavare direttamente il passo iterativo che sarà:

$$x_{i+1} = x_i - \frac{x_i^2 - x_0}{2x_i} = \frac{1}{2}(x_i + \frac{x_0}{x_i})$$

La massima precisione possibile è stata ottenuta utilizzando come tolleranza la precisione di macchina contenuta nella variabile eps di Matlab, infatti sarebbe inutile usare un valore di tolleranza inferiore in quanto sotto tale soglia non avremo alcuna garanzia che la tolleranza venga rispettata.

Function richesta:

```
   function rad=radice(x)
%rad=radice(x)
%Input: x, numero di cui si vuole conoscere la radice
%Output: rad, radice quadrata del numero in input
%Restituisce la radice quadrata del numero passato in input.
if x==0
    rad=0;
    return;
end
if x<0, error("La radice quadrata di un numero negativo non e' un numero reale!"), end
x0=x;
for i=0:1000
    rad=0.5*(x+x0/x); %passo iterativo
    if(abs(rad-x)<eps*(1+abs(x))) %controllo di aver raggiunto la massima precisione possibile
        break;
    end
    x=rad;
end
return;
```

Tabella di confronto per i valori equispaziati logaritmicamente

Valore	radice	sqrt
1e-10	9.999999999999999e-06	1.000000000000000e-05
1e-9	3.162277660168380e-05	3.162277660168380e-05
1e-8	9.999999999999999e-05	1.000000000000000e-04
1e-7	3.162277660168380e-04	3.162277660168380e-04
1e-6	1.000000000000000e-03	1.000000000000000e-03
1e-5	0.003162277660168	0.003162277660168
1e-4	0.010000000000000	0.010000000000000
1e-3	0.031622776601684	0.031622776601684
1e-2	0.100000000000000	0.100000000000000
1e-1	0.316227766016838	0.316227766016838
1e0	1	1
1e1	3.162277660168379	3.162277660168380
1e2	10	10
1e3	31.622776601683793	31.622776601683793
1e4	100	100
1e5	3.162277660168379e+02	3.162277660168380e+02
1e6	1000	1000
1e7	3.162277660168379e+03	3.162277660168379e+03
1e8	10000	10000
1e9	3.162277660168379e+04	3.162277660168379e+04
1e10	100000	100000

Le differenze sono dovute all'uso dell'aritetica finita per rappresentare i numeri reali che necessita il ricorso ad approssimazioni per i valori che non appartengono ad i numeri di macchina.

Esercizio 5
Scrivere function Matlab distinte che implementino efficientemente i seguenti metodi per la ricerca degli zeri di una funzione f(x):

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

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

Per tutti i metodi, utilizzare come criterio di arresto $|x_{n+1} - x_n| \leq tol(1 + |x_n|)$, essendo tol una opportuna tolleranza specificata in ingresso. Curare particolarmente la robustezza del codice.

Soluzione

Metodo di Newton

```
function [xfin, passi]=newton(x0, func, der, tol, max)
%[xfin, passi]=newton(x0, func, der, tol, max)
%Output
%xfin: approsimazione della radice ottenuta, passi: passi effetutati dal metodo prima di
    raggiungere la tolleranza desiderata.
%Input
%x0: punto di partenza del metodo, func: function handler contenente una funzione(sufficientemente
    regolare) di cui si vuole conoscere la radice,
%der: derivata prima di tale funzione, (opzionali) tol: tolleranza desiderata(impostata di default
    sulla precisone di macchina) , max=
%numero massimo di passi effetuati dal metodo(di default 1000)
%
%Calcola un'approssimazione della radice della funzione passata in input con il metodo di Newton
%usando come punto d'innesco il valore passato in input
passi=0;
if ~exist("max", "var"), max=1000; end %gestico il caso di input mancanti
if ~exist("tol", "var"), tol=eps; end
for i=1:max
    x1=x0;
    f0=func(x0);
    d0=der(x0);
    if(d0==0), disp("Derivata prima uguale a 0, usare un nuovo punto d'innesco o una funzione piu'
        regolare!"), xfin=x0; break,end
    passi=passi+1;
    xfin=x0-f0/d0; %passo iterativo del metodo di Newton
    if(abs(xfin-x0)<tol*(1+abs(x0))) %controllo se la tolleranza e' rispettata
        break;
    end
    x0=xfin;
end
if abs(xfin-x1)>tol*(1+abs(x0)), disp("il metodo non e' convergente!"), end
```

Tale script converge quadraticamente ad una radice semplice(ovvero la cui molteplicità è uguale ad 1) di una funzione sufficientemente regolare in un suo opportuno intorno(convergenza locale). Invece essa converge (localmente) linearmente verso radici multiple(molteplicità maggiore di 1). Inoltre il costo per iterazione è di due valutazioni funziali.

Metodo delle secanti

```
 function [xfin,passi]=secanti(x0, x1, func, tol, max)
%[xfin, passi]=secanti(x0, x1, func, tol, max)
%Output
%xfin: radice ottenuta, passi: passi effetutati dal metodo prima di raggiungere la tolleranza
    desiderata.
%Input
%x0: punto d'innesco del metodo, func: funzione di cui si vuole conoscere la radice,
%x1: punto calcolato con un iterazione del metodo di Newton da x0, (opzionali) tol: tolleranza
    desiderata(impostata di default sulla precisone di macchina) , max=
%numero massimo di passi effetuati dal metodo(di default 1000)
%Calcola la radice della funzione passata in input con il metodo delle
%secanti usando come punto d'innesco il valore passato in input
passi=0;
if ~exist("max", "var"), max=1000; end %gestico il caso di input mancanti
if ~exist("tol", "var"), tol=eps; end
f=func(x0);
xfin=x1;
for i=1:max
    passi=passi+1;
    if(abs(xfin-x0)<tol*(1+abs(x0))) %controllo se la tolleranza e' rispettata
        break;
    end
    f0=f;
    f=func(xfin);
    if(f0==f), disp("Precisione massima raggiunta impossibile proseguire!"),break,end
    x1=(f*x0-f0*xfin)/(f-f0); %passo iterativo del metodo delle Secanti
    x0=xfin;
    xfin=x1;
end
if abs(xfin-x1)>tol*(1+abs(x0)), disp("il metodo non e' convergente!"), end
```

Questa Function converge (localmente) verso radici semplici con ordine di convergenza $p = \frac{\sqrt{5}+1}{2}$ mentre linearmente verso radici multiple ed ha un costo per iterazione di una sola valutazione funzionale e non richiede di conoscere la derivata prima della funzione come nel metodo di Newton.

Metodo di Steffensen

```
    function [xfin,passi]=steffensen(x0, func, tol, max)
%[xfin,passi]=steffensen(x0, func, tol, max)
%Input: x0 punto d'innesco del metodo, func function handler della funzione di cui si vuole
%conoscere la radice, (opzionali) tol tolleranza desiderat, max massimo
%numero di iterazioni.
%Output: xfin radice trovata dal metodo, passi passi necessari per trovare
%la soluzione.
%Cerca la radice della funzione passata in input con il metodo di
%Steffensen.
passi=0;
if ~exist("max", "var"), max=100; end %gestico il caso di input mancanti
if ~exist("tol", "var"), tol=eps; end
for i=0:max
    x1=x0;
    f0=func(x0);
    f1=func(x0+f0)-f0;
    if f1==0 %evito di dividere per 0
        disp("Non si puo' dividere per 0, radice non ulteriormente approssimabile!"), break
    end
    f=(f0^2)/f1; %passo iterativo del metodo di Steffensen
    xfin=x0-f;
    passi=passi+1;
    if(abs(xfin-x0)<tol*(1+abs(x0))) %controllo se la tolleranza e' rispettata
        break;
    end
    x0=xfin;
end
if abs(xfin-x1)>tol*(1+abs(x0)), disp("il metodo non e' convergente!"), end
```

Questo script ha un costo di 2 valutazioni funzionali per iterazione e non richiede di conoscere la derivata prima della funzione come invece accade per il metodo di Newton.

Esercizio 6
Utilizzare le function del precedente esercizio per determinare una approssimazione della radice della funzione

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

partendo da $x_0 = 1$ (e $x_1 = 0.99$ per il metodo delle secanti). Tabulare i risultati, in modo da confrontare le iterazioni richieste da ciascun metodo. Commentare il relativo costo computazionale, in termini di valutazioni funzionali richieste.

Soluzione
I risultati sono riportati di seguito:

Newton		**Secanti**		**Steffensen**		/ / /
Iterazioni	**Radice**	**Iterazioni**	**Radice**	**Iterazioni**	**Radice**	tol.
3	0.594611646360541	4	0.594618477671711	4	0.594611681141925	1e-3
4	0.594611644056836	6	0.594611644056842	5	0.594611644056837	1e-6
5	0.594611644056836	7	0.594611644056836	6	0.594611644056836	1e-9
5	0.594611644056836	7	0.594611644056836	6	0.594611644056836	1e-12

Per valutare il costo computazionale di ciascun algoritmo è necessario conoscere le valutazioni funzionali che ogni metodo richiede per iterazione: 2 per il metodo di Newton (il calcolo della funzione e della derivata nel punto) e di Steffensen(il calcolo della funzione e di $f(x_n + f(x_n))$ mentre per il metodo delle Secanti ne è necessaria una soltanto.
Per questo si può affermare che in questo caso il metodo che richiede meno valutazioni funzionali in generale è il metodo delle secanti (che ne richiede 4-6-7-7 se non si considera la prima iterazione necessaria per trovare x_1), mentre gli altri due metodi, pur con meno iterazioni, ne richiedono un numero maggiore(6-8-10-10 per Newton e 8-10-12-12 per Steffensen).
Inoltre per il metodo di Newton è necessario calcolare la derivata prima che risulta essere:

$$f'(x) = 1 + \frac{\pi}{2}sin(\frac{\pi}{2}x)$$

Esercizio 7
Utilizzare le function del precedente esercizio per determinare una approssimazione della radice della funzione

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

partendo da $x_0 = 1$ (e $x_1 = 0.99$ per il metodo delle secanti). Tabulare i risultati, in modo da confrontare le iterazioni richieste da ciascun metodo. Commentare i risultati ottenuti.

Soluzione
I risultati sono riportati di seguito:

Newton		**Secanti**		**Steffensen**		/ / /
Iterazioni	**Radice**	**Iterazioni**	**Radice**	**Iterazioni**	**Radice**	tol.
13	0.596947934307877	17	0.599143722778773	18	0.597396552671652	1e-3
30	0.594614017981881	42	0.594615663476623	35	0.594614370677192	1e-6
47	0.594611646466276	66	0.594611648768801	36	0.594613282745380	1e-9
64	0.594611644059281	91	0.594611644061006	36	0.594613282745380	1e-12

La radice desiderata, seppur identica a quella dello scorso esercizio, non è semplice in quanto ha molteplicità 3 il che rende l'ordine di convergenza dei metodi di Newton e delle Secanti lineare e non più rispettivamente quadratico e $\frac{\sqrt{5}+1}{2}$ come nel caso di radici semplici.
Inoltre possiamo notare come il metodo di Steffensen, una volta che l'approssimazione si avvicina sufficientemnte alla radice, si ferma a prescindere dalla tolleranza specificata in quanto $f(x_i)$ diventa molto vicino a 0 il che rischia di portare a 0 anche il denominatore del passo iterativo del metodo, il che viene rilevato dal controllo che interrompe l'esecuzione del metodo e restituisce l'approssimazione ottenuta.

Esercizio 8
Scrivere una function Matlab: function $x = mialu(A, b)$
che, data in ingresso una matrice A ed un vettore b, calcoli la soluzione del sistema lineare Ax = b con il metodo di fattorizzazione LU con pivoting parziale. Curare particolarmente la scrittura e l'efficienza della function, e validarla su due esempi non banali, generati casualmente, di cui sia nota la soluzione.

Soluzione

```
   function x = mialu(A,b)
%x=mialu(A,b)
%Input
%A: Matrice quadrata nonsingolare(det(A)!=0) del sistema da risolvere
%b: Vettore colonna dei termini noti
%Output
%x: Soluzione del sistema lineare
%Risolve il sistema lineare Ax=b, passato in input, con la fattorizzazione LU con pivoting parziale
if(size(A,1)~=size(A,2)), error("Matrice non quadrata!"), end
if(size(b,2)>1),error("Inserire un vettore colonna come vettore dei termini noti!"),end
if(size(A,1)~=size(b,1)), error("Dimensione del vettore dei termini noti non compatibile con la
    matrice!"), end

[LU,p]=LUpivoting(A); %Fattorizzo la matrice A con la fattorizzazione LU con pivoting parziale
x=b;
x=x(p);     %scambio le posizioni degli elementi del vettore dei termini noti in accordo con
    quelli effettuati per il pivoting
x=LUsolver(LU, x); %Risolvo il sistema lineare
return
end

function [LU,p]= LUpivoting(A)
% [LU,p]=LUpivoting(A)
% Restiusce la fattorizzazione LU della matrice nonsingolare A con pivoting
% parziale di cui teniamo traccia nel vettore p
LU=A;
n=size(LU);
p=1:n; %genero un vettore contenente i numeri da 1 ad n
for i=1:n-1
    [ma,k]=max(abs(LU(i:n,i))); %individuo il valore massimo del vettore che prendo in
        considerazione
    if(ma==0), error("Matrice singolare!"), end
    k=k+i-1;
    if k>i
        LU([i k],:)=LU([k i], :); %scambio le righe
        p([i k])=p([k i]); %tengo traccia dello scambio nel vettore delle permutazioni
    end
    LU(i+1:n,i)=LU(i+1:n,i)/LU(i,i);
    LU(i+1:n,i+1:n)= LU(i+1:n,i+1:n)- LU(i+1:n,i)*LU(i, i+1:n);
end
return;
end

function x=LUsolver(LU,b)
%x=LUsolver(LU,b)
% Risolve il sistema lineare LUx=b gia' decomposto con la
%fattorizzazione LU
x=b;
n=size(LU);
```

```
for i=1:n-1
    x(i+1:n)=x(i+1:n)-LU(i+1:n,i)*x(i); %risoluzione fattore L per colonne
end
for i=n:-1:1
    x(i)=x(i)/LU(i,i);
    x(1:i-1)=x(1:i-1)-LU(1:i-1,i)*x(i); %risoluzione fattore U per colonne
end
return;
end
```

La complessità di tale script è di circa $\frac{2}{3}n^3$ per la funzione di scomposizione LU con pivoting parziale e di circa n^2 per risolvere il sistema lineare precedentemente scomposto. Per validare lo script ho generato casualmente due matrici con la function randi di Matlab e due vettori dei termini noti con lo stesso procedimento e li ho successivamente provati a risolvere per verificare il corretto funzionamento salvando la soluzione ottenuta nella variabile x:

Primo test:

$$A = \begin{bmatrix} 8 & 10 & 9 \\ 6 & 7 & 9 \\ 5 & 7 & 6 \end{bmatrix}, b = \begin{bmatrix} 7 \\ 2 \\ 1 \end{bmatrix}$$

Soluzione:

$$x = \begin{bmatrix} 7.6000 \\ -3.4000 \\ -2.2000 \end{bmatrix}$$

In questo caso la soluzione è corretta (verificabile controllando che $Ax - b = 0$), le piccole differenze sono dovute all'errore di rappresentazione causato all'utilizzo di un aritmetica finita moltiplicato per il fattore di condizionamento della matrice A che è dell'ordine delle decine.

Secondo test:

$$A = \begin{bmatrix} 6 & 14 & 9 & 4 \\ 1 & 7 & 8 & 10 \\ 2 & 20 & 6 & 9 \\ 17 & 1 & 16 & 13 \end{bmatrix}, b = \begin{bmatrix} 15 \\ 16 \\ 6 \\ 14 \end{bmatrix}$$

Soluzione:

$$x = \begin{bmatrix} 2.048015873015874 \\ 1.913095238095239 \\ -4.188888888888891 \\ 3.407142857142858 \end{bmatrix}$$

Come prima calcolando Ax-b si nota che il residuo è inferiore alla precisione di macchina moltiplicata per il fattore di condizionamento di A, per cui possiamo ritenere soddisfacente la soluzione.

Esercizio 9
Scrivere una function Matlab, function

$$x = mialdl(A, b)$$

che, dati in ingresso una matrice sdp A ed un vettore b, calcoli la soluzione del corrispondente sistema lineare utilizzando la fattorizzazione LDL^T. Curare particolarmente la scrittura e l'efficienza della function, e validarla su due esempi non banali, generati casualmente, di cui sia nota la soluzione.

Soluzione

```
function x=mialdl(A,b)
%x=mialdl(A,b)
%INPUT
%A: Matrice simmetrica definita positiva(sdp), b: Vettore dei termini noti
%OUTPUT
%x: Soluzione del sistema lineare Ax=b
%Risolve il sistema lineare passato in input grazie alla fattorizzazione LDLt.
if(size(A,1)~=size(A,2)), error("Matrice non quadrata!"), end
if(size(A,1)~=size(b)), error("Dimensione del vettore dei termini noti non compatibile con la
    matrice!"), end
LDLt=LDLtdecompose(A); %decompongo la matrice tramite fattorizzazione LDLt
x=trinfdiaguni(LDLt,b); %Risolvo Lx=b
x=diagonal(LDLt,x); %Risolvo Dx=b dove b e' la soluzione x ricavata in precedenza
x=trsupdiaguni(LDLt',x); %Risolvo L'x=b dove b e' la soluzione x ricavata in precedenza
return;
end

function LDLt = LDLtdecompose(A)
%LDLt = LDLtdecompose(A)
%decompone la matrice A simmetrica definita positiva tramite
%fattorizzazione LDLt dove L e' una matrice triangolare inferiore a
%diagonale unitaria e D una matrice diagonale
n= size(A);
LDLt=A;
if(LDLt(1,1)<=0), error('Matrice non sdp!'),end
LDLt(2:n,1)=LDLt(2:n,1)/LDLt(1,1);
for i=2:n
v=(LDLt(i,1:i-1)').*diag(LDLt(1:i-1,1:i-1));
LDLt(i,i)=LDLt(i,i)-LDLt(i,1:(i-1))*v;
if(LDLt(i,i)<=0),error('Matrice non SDP!'),end
LDLt(i+1:n,i)=(LDLt(i+1:n,i)-LDLt(i+1:n,1:i-1)*v)/LDLt(i,i);
end
end

function b = trinfdiaguni(A,b)
%risolve il sistema lineare Ax=b con A matrice triangolare inferiore a diagonale unitaria
n=size(A,1);
for i=1:n
b(i)=b(i)-A(i,1:i-1)*b(1:i-1);
end
end

function x = diagonal(D, b)
%x = diagonal(D, b)
%risolve il sistema lineare Dx=b con D matrice diagonale
```

```
n=size(D);
x=b;
for i=1:n
    x(i)=x(i)./D(i,i);
end
end

function x = trsupdiaguni(A,b)
%x = trsupdiaguni(A,b)
%risolve il sistema lineare Ax=b con A matrice triangolare superiore a diagonale unitaria
n=size(A);
x=b;
for i=n:-1:1
x(i)=x(i)-A(i,i+1:n)*x(i+1:n);
end
end
```

La complessità di tale script è di circa $\frac{1}{3}n^3$ per la funzione di scomposizione LDL^T e sempre di circa n^2 per risolvere il sistema lineare precedentemnte scomposto. Per validare lo script ho generato casualmente due matrici come nel precedente esercizio moltiplicando però la matrice ottenuta per la traposta di se stessa in modo da ottenere così una matrice quasi sicuramente SDP(potrebbe anche generare una matrice che è simmetrica ma non definita positiva, tale problema viene comunque rilevato dai controlli dello script):

Primo test:

$$A = \begin{bmatrix} 9 & 8 & 10 \\ 8 & 9 & 10 \\ 10 & 1 & 12 \end{bmatrix}, b = \begin{bmatrix} 2 \\ 6 \\ 3 \end{bmatrix}$$

Soluzione:

$$x = \begin{bmatrix} 2.5000 \\ 6.5000 \\ -7.2500 \end{bmatrix}$$

La soluzione trovata è corretta (verificabile come prima controllando che $Ax - b = 0$).

Secondo test:

$$A = \begin{bmatrix} 13 & 9 & 9 & 11 \\ 9 & 7 & 6 & 7 \\ 9 & 6 & 7 & 8 \\ 11 & 7 & 8 & 10 \end{bmatrix}, b = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 2 \end{bmatrix}$$

Soluzione:

$$x = \begin{bmatrix} -11.0000 \\ 7.0000 \\ -2.0000 \\ 9.0000 \end{bmatrix}$$

Anche in questo caso la soluzione risulta essere corrtetta.

Esercizio 10
Data la function Matlab

```
function [A,b] = linsis(n,k,simme)
%
% [A,b] = linsis(n,k,simme) Crea una matrice A nxn ed un termine noto b,
% in modo che la soluzione del sistema lineare
% A*x=b sia x = [1,2,...,n]'.
% k 'e un parametro ausiliario.
% simme, se specificato, crea una matrice
% simmetrica e definita positiva.
%
sigma = 10^(-2*(1-k))/n;
rng(0);
[q1,r1] = qr(rand(n));
if nargin==3
q2 = q1';
else
[q2,r1] = qr(rand(n));
end
A = q1*diag([sigma 2/n:1/n:1])*q2;
x = [1:n]';
b = A*x;
return
```

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

Soluzione
Di seguito sono riportati i vettori soluzione ottenuti grazie alla function *mialdlt*:

$$x_1 = \begin{bmatrix} 1.000000000000000 \\ 1.999999999999999 \\ 3.000000000000001 \\ 3.999999999999997 \\ 5.000000000000007 \\ 5.999999999999998 \\ 6.999999999999997 \\ 7.999999999999997 \\ 8.999999999999998 \\ 10.000000000000000 \end{bmatrix}$$

$$x_2 = \begin{bmatrix} -1057.091101303854 \\ 608.000000000003 \\ -960.000000000006 \\ -852.000000000005 \\ 116.000000000000 \\ 240.000000000002 \\ -1380.000000000007 \\ -3136.000000000013 \\ 1268.000000000006 \\ 464.000000000002 \end{bmatrix}$$

Il motivo per cui la soluzione è cosi diversa rispetto a quella attessa(risultando così poco accurata) è

riconducibile all'elevato numero di condizionamento della matrice di partenza, infatti il numero di condiziozionamento a 2 della seconda matrice(calcolato con la function cond di Matlab) risulta essere circa $7.119787475990509 * 10^{18}$ che va moltiplicato per la precisione di macchina (ovvero il massimo errore relativo dovuto alla rappresentazione in aritmetica finita dei numeri reali) come espresso nella formula:

$$\frac{||\Delta x||}{||x||} \leq k(A) \left(\frac{||\Delta A||}{||A||} + \frac{||\Delta b||}{||b||} \right)$$

Dove $k(A)$ è il numero di condizionamento di A e $\frac{||\Delta A||}{||A||}$, $\frac{||\Delta b||}{||b||}$ ed $\frac{||\Delta x||}{||x||}$ rappresentano il rapporto fra la norma (indotta nel caso di A) della perturbazione rispettivamente di A, b ed x e quella delli stessi. Tale rapporto è assimilabile ad una sorta di "errore relativo" del vettore o matrice stesso. Usando quindi come "errore relativo" di A e b la precisione di macchina è possibile stimare che per x tale "errore relativo" massimo risulta essere dell' ordine di 10^3, cosa che rispecchia la pertubazione ottenuta dalla seconda soluzione (tale discorso può essere applicato anche alla prima soluzione tenendo però di conto che il numero di condizionamento della prima matrice risulta essere notevolmente inferiore rispetto alla seconda risultando essere circa 10).

Esercizio 11
Risolvere, utilizzando la function mialdlt, i sistemi lineari generati da $[A, b] = linsis(10, 1, 1)$ e $[A, b] = linsis(10, 10, 1)$. Commentare l'accuratezza dei risultati ottenuti, dandone spiegazione esaustiva.

Soluzione
Di seguito sono riportati i vettori soluzione ottenuti grazie alla function *mialu*:

$$x_1 = \begin{bmatrix} 0.999999999999998 \\ 1.999999999999996 \\ 2.999999999999999 \\ 3.999999999999998 \\ 4.999999999999997 \\ 6.000000000000000 \\ 6.999999999999998 \\ 7.999999999999996 \\ 8.999999999999996 \\ 9.999999999999996 \end{bmatrix}$$

La seconda soluzione risulta invece impossibile da calcolare con la function mialdl a causa dell'elevato numero di condizione di A (circa 10^{18}) che rende del tutto mal condizionata la matrice(e poco accurata un eventuale soluzione del sistema lineare) al punto da non farla più risultare definita positiva al controllo del secondo elemento sulla diagonale alla seconda iterazione del metodo. La prima soluzione risulta essere invece sufficientemente accurata essendo il numero di condizionamento della matrice A uguale a 10.

Esercizio 12
Scrivere una function Matlab,

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

che, data in ingresso la matrice A m × n, con m m≥n = rank(A), ed un vettore b di lunghezza m, calcoli la soluzione del sistema lineare Ax = b nel senso dei minimi quadrati e, inoltre, la norma, nr, del corrispondente vettore residuo. Curare particolarmente la scrittura e l'efficienza della function. Validare la function miaqr su due esempi non banali, generati casualmente, confrontando la soluzione ottenuta con quella calcolata con l'operatore Matlab \

Soluzione

```
function [x,nr] = miaqr(A,b)
%[x,nr] = miaqr(A,b)
%Input: A matrice sovradimensionata a rango massimo(non quadrata, vedasi function mialu
    computazionalmente piu' efficiente),
%b vettore colonna dei termini noti.
%Output: x soluzione del sistema lineare nel senso dei minimi quadrati,
% nr: norma 2 del vettore residuo.
%Calcola la soluzione, nel senso dei minimi quadrati, del sistema lineare
%sovradeterminato passato in input e restiuisce anche la norma due del
%vettore dei residui (minimizzata grazie al metodo dei minimi quadrati).
[m,n] = size(A);
k=size(b,1);
if (n>=m),error("Sistema non sovradeterminato, conviene usare un'altro algoritmo piu' efficiente
    (vedasi function mialu)!"),end
if(m~=k), error("Le lunghezze della matrice e del vettore dei termini noti non coincidono!"), end
if(size(b,2)>1),error("Inserire un vettore colonna come vettore dei termini noti!"),end
QR=QRdecompose(A); %decompongo
[x,nr]=QRsolver(QR,b);
return;
end


function QR=QRdecompose(A)
%QR=QRdecompose(A)
%Input: A matrice sovradimensionata a rango massimo
%Output: QR: matrice fattorizzata QR
%Decompone A matrice sovradimensionata nella sua fattorizzazione QR dove Q e' una matrice ortogonale
% e R una matrice composta nelle prime n righe da una matrice triangolare superiore di dimensione
    nXn ed il resto delle righe nullo
[m,n] = size(A);
norma2=norm(A);
QR=A;
for i = 1 : n
alpha = norm(QR(i : m, i));
if abs(alpha)<=eps*norma2
error('Matrice non a rango massimo!');
end
if QR(i,i)>=0
alpha=-alpha;
end
v1=QR(i,i)-alpha;
QR(i,i)=alpha;
QR(i+1:m,i)=QR(i+1:m,i)/v1;
beta =-v1/alpha;
QR(i:m,i+1:n)=QR(i: m,i+1:n)-(beta*[1;QR(i+1:m,i)])*([1 QR(i+1:m,i)']*QR(i:m,i+1:n));
end
end
```

```
function [x,nr] = QRsolver(QR,b)
%[x,nr] = QRsolver(QR,b)
%Input: QR: matrice precedentemente fattorizzata QR, b: vettore dei termini
%noti
%Output: x: soluzione del sistema lineare in input
%Risolve il sistema lineare con la scomposizione QR della matrice
%precedentemnte ottenuta.
x=b;
[m,n] = size(QR);
for i=1:n
v=[1;QR(i+1:m,i)];
x(i:m)=x(i:m)-((2*(v*v'))/(v'*v))*x(i:m); %Non necessito di "costruirmi" la matrice Q ma utilizzo
    solo i vettori v
end
x(1:n)=trsup(QR(1:n , 1 : n), x(1 : n));
nr=norm(x(n+1:m)); %mi ricavo la norma euclidea del vettore residuo
x=x(1:n);
end

function [b] = trsup(A,b)
%[b] = trsup(A,b)
%Input: A matrice quadrata triangolare superiore, b vettore dei termini
%noti
%Output: x: soluzione del sistema lineare in input
%Risolve il sistema lineare Ax=b passato in input dove A e' una matrice triangolare superiore.
n=size(A,1);
for i=n:-1:1
    if(n>1)
    b(i)=b(i)-A(i,i+1:n)*b(i+1:n);
    end
    b(i)=b(i)/A(i,i);
end
end
```

La complessità di tale script è di circa $\frac{2}{3}n^2(3m-n)$ per la scomposizione QR dove m è il numero delle righe ed n il numero di colonne di A. In termini di efficienza nell'utilizzo della memoria è utile notare come non si sia ricorso alla memorizzazione della matrice Q della scomposizione QR(che andrebbe poi moltiplicata per il vettore b visto l'ortogonalità di Q) in quanto abbiamo moltiplicato direttamente il vettore b per Q sfruttando la definizione di matrice di Householder $H^i = I_i - 2\frac{v_i v_i^T}{v_i^T v_i}$ da cui segue $H^i x = x - 2\frac{v_i v_i^T}{v_i^T v_i}x$ tenendo conto che:

$$H_i = \left(\begin{array}{c|c} I_i & \\ \hline & H^i \end{array}\right)$$

è stato necessario semplicemente moltiplicare il sotto-vettore di lunghezza i (con i che va da 1 fino al numero di colonne di A) dal basso per la matrice H^i sfruttando la scomposizione sopra per ottenere il risultato desiderato.

Primo test:

$$A = \begin{bmatrix} 7 & 1 \\ 4 & 5 \\ 10 & 4 \end{bmatrix}, b = \begin{bmatrix} 1 \\ 3 \\ 5 \end{bmatrix}$$

Soluzione:

$$x = \begin{bmatrix} 0.199098730028677 \\ 0.539532978287587 \end{bmatrix}$$

$$nr = 1.356097492685599$$

La soluzione è corretta in quanto identica a quella generata utilizzando l'operatore \. **Secondo test:**

$$A = \begin{bmatrix} 3 & 9 & 3 \\ 7 & 6 & 8 \\ 5 & 6 & 8 \\ 4 & 10 & 4 \end{bmatrix}, b = \begin{bmatrix} 3 \\ 1 \\ 1 \\ 3 \end{bmatrix}$$

Soluzione:

$$x = \begin{bmatrix} -0.011722565939433 \\ 0.374308042982742 \\ -0.147671768153696 \end{bmatrix}$$

$$nr = 0.153118032507170$$

Anche in questo caso la soluzione è corretta in quanto identica a quella generata con l'operatore \.

Esercizio 13

Utilizzare la function miaqr per risolvere, nel senso dei minimi quadrati, i sistemi lineari sovra-determinati

$$Ax = b, (D * A)x = (D * b)$$

definiti dai seguenti dati:

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

Commentare i risultati ottenuti.

Soluzione

Di seguito è riportata la soluzione del sistema lineare sovra-determinato $Ax = b$ calcolata con la function *miaqr*:

$$x = \begin{bmatrix} 3.000000000000008 \\ 5.800000000000001 \\ -2.500000000000009 \end{bmatrix}$$

$$nr = 1.264911064067357$$

Possiamo notare al solito una piccola inaccuratezza dovuta alla rappresentazione dei numeri reali non di macchina (cosa riscontrabile anche nella soluzione ottenuta con l'operatore \), al netto di ciò la soluzione fornita dalla function resta comunque soddisfacente (è interessante notare inoltre come la somma degli elementi del vettore residuo sia 0).

Ora riporto invece la soluzione ottenuta sempre con la function *miaqr* del sistema $(D * A)x = (D * b)$:

$$x = \begin{bmatrix} -0.602569986232221 \\ 4.701698026617711 \\ 1.758375401560359 \end{bmatrix}$$

$$nr = 3.735151112342432$$

Allo stesso modo le soluzioni sono leggermente discostanti rispetto a quelle ottenute con l'operatore \ per l'approssimazione dei numeri reali in numeri di macchina, in questo caso però la norma 2 del vettore residuo è maggiore rispetto al caso precedente.

Esercizio 14
Scrivere una function Matlab,

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

che implementi efficientemente il metodo di Newton per risolvere sistemi di equazioni non-lineari. Curare particolarmente il criterio di arresto, che deve essere analogo a quello usato nel caso scalare. La seconda variabile, se specificata, ritorna il numero di iterazioni eseguite. Prevedere opportuni valori di default per gli ultimi due parametri di ingresso.

Soluzione

Per evitare di avere più function con lo stesso nome, nella cartella allegata con i codici Matlab, questa function è chiamata newton2

```
function [x,nit] = newton(fun,jacobian,x0,tol,maxit)
%[x,nit] = newton(fun,jacobian,x0,tol,max)
%Input: fun vettore di handler di funzioni nonlineari di cui si vuole trovare le radici,
%jacobian: matrice di handler di funzioni contenenti il giacobiano di fun, x0 vettore d'innesco
    del metodo,
%(opzionali) tol tolleranza desiderata(di default precisione di macchina),
%max massimo numero di iterazioni del metodo(di default 1000)
%Output
%x: Vettore contente le radici del sistema non lineare, nit numero di
%iterazioni compiute dal metodo per trovare la radice
%Trova il vettore radice del sistema di equzioni non lineari passati in
%input col metodo di Newton partendo dal vettore d'innesco passato in input
nit=0;
if ~exist("maxit", "var"), maxit=1000; end %gestico il caso di input mancanti
if ~exist("tol", "var"), tol=eps; end
x=x0;
for i=1:maxit
    f=-fun(x);
    j=jacobian(x);
    dx=mialu(j,f); %uso la function mialu vista in precedenza per ricavarmi il vettore della
        differenza dx
    x=x+dx;
    nit=nit+1;
    if(norm(dx)<=tol*(1+norm(x-dx))) %controllo se la tolleranza e' rispettata
        break;
    end
end
if (norm(dx)>tol*(1+norm(x-dx))), disp('il metodo non e' convergente!'), end
```

Tale script converge quadraticamente ad una radice in un opportuno intorno di essa se la funzione ha derivate seconde continue, con convergenza locale come nel caso scalare precedentemente illustrato.

Esercizio 15
Usare la function del precedente esercizio per risolvere, a partire dal vettore iniziale nullo, i seguenti sistemi non-lineari, utilizzando tolleranze $tol = 1e-3, 1e-8, 1e-13$:

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

Tabulare i risultati ottenuti, commentandone l'accuratezza.

Soluzione
Di seguito sono riportatile radici del primo sistema nonlineare ottenute grazie alla function *newton* ed il numero di iterazioni necessario per ricavarle con la tolleranza desiderata:

Tolleranza	Radice(x_1)	Radice(x_2)	Iterazioni
10^{-3}	1.000000007127784	2	7
10^{-8}	1	2	8
10^{-13}	1	2	9

Questi sono invece i risultati dell'applicazione della function al secondo sistema nonlineare:

Tolleranza	Radice(x_1)	Radice(x_2)	Radice(x_3)	Iterazioni
10^{-3}	0.999984513614475	0.999983797366917	1.000015844509304	9
10^{-8}	0.999999999999486	0.999999999999486	1.000000000000514	12
10^{-13}	1	1	1	14

Per verificarne la correttezza e l'accuratezza è sufficiente calcolare il sistema nonlineare nel vettore restiuito e osservare che in entrambi i casi la funzione calcolata con l'approssimazione calcolata con tolleranza minore assume valori nulli(va inoltre notato come le soluzioni con tolleranza maggiore si discostino dalla soluzione con un valore congruo alla tolleranza specificata.

Esercizio 16
Costruire una function, lagrange.m, avente la stessa sintassi della function spline di Matlab, che implementi, in modo vettoriale, la forma di Lagrange del polinomio interpolante una funzione.

Soluzione

```
function YQ=lagrange(X,Y,XQ)
% YQ=lagrange(X,Y,XQ)
%
%Input
%X: Vettore colonna contenete le ascisse d'interpolazione,
%Y: Vettore colonna contenete i valori della funzione nelle ascisse d'interpolazione
%XQ: Vettore colonna contenente le ascisse in cui vogliamo approssimare la funzione
%Output
%YQ: Valori approssimati della funzione con il polinomio interpolante in
%forma di Lagrange
%
%Calcola i valori approssimati della funzione(di cui conosciamo i valori Y che assume nelle
    ascisse X) calcolati attraverso il polinomio interpolante in
%forma di Lagrange nelle ascisse XQ.
%
if(length(X)~=length(Y)), error("Dati in ingresso errati!"),end
if (length(X) ~= length(unique(X))), error("Ascisse d'interpolazione non distinte!"),end %uso la
    function unique che restuisce un vettore contenente i valori senza ripetizioni di X
if(isempty(XQ)), error("Il vettore contenente le ascisse in cui interpolare la funzione e'
    vuoto!"),end
if(size(X,2)>1||size(Y,2)>1||size(XQ,2)>1),error("Inserire vettori colonna!"),end
n=size(X,1);
L=ones(size(XQ,1),n);
   for i=1:n
      for j=1:n
         if (i~=j)
            L(:,i)=L(:,i).*((XQ-X(j))/(X(i)-X(j))); %calcolo i polinomi di base di lagrange Lin(x)
         end
      end
   end
   YQ=0;
   for i=1:n
      YQ=YQ+Y(i).*L(:,i);  %calcolo la sommatoria dei prodotti fi*Lin(x) (con i=0,...,n)
   end
end
```

Per questa function è stato sufficiente calcolare i polinomi di base di Lagrange $Lin(x) = \prod_{j=0, j\neq i}^{n} \frac{x-x_j}{x_i-x_j}$ per poi ricavarmi direttamente i valori (che ho salvato nel vettore YQ) tramite la sommatoria $p(x) = \sum_{i=0}^{n} fi * Lin(x)$ (dove n è il numero di ascisse d'interpolazione meno uno) dove i valori fi sono contenuti nel vettore Y e le ascisse d'interpolazione nel vettore X.

Esercizio 17
Costruire una function, newton.m, avente la stessa sintassi della function spline di Matlab, che implementi, in modo vettoriale, la forma di Newton del polinomio interpolante una funzione.

Soluzione

Per evitare di avere più function con lo stesso nome, nella cartella allegata con i codici Matlab, questa function è chiamata newtonint

```
function YQ=newton(X,Y,XQ)
% YQ=newton(X,Y,XQ)
%
%Input
%X: Vettore colonna contenete le ascisse d'interpolazione che devono essere distinte,
%Y: Vettore colonna contenete i valori della funzione nelle ascisse d'interpolazion
%XQ: Vettore contente le ascisse in cui vogliamo approssimare la funzione
%Output
%YQ: Valori approssimati della funzione con il polinomio interpolante in
%forma di Newton
%
%Calcola i valori approssimati della funzione(di cui conosciamo i valori Y che assume nelle
    ascisse X) calcolati attraverso il polinomio interpolante in
%forma di Newton nelle ascisse XQ.
%
if(length(X)~=length(Y)), error("Numero di ascisse d'interpolazione e di valori della funzione non
    uguale!"),end
if (length(X) ~= length(unique(X))), error("Ascisse d'interpolazione non distinte!"),end %uso la
    function unique che restuisce un vettore contenente i valori senza ripetizioni di X
if(isempty(XQ)), error("Il vettore contenente le ascisse in cui interpolare la funzione e'
    vuoto!"),end
if(size(X,2)>1||size(Y,2)>1),error("Inserire vettori colonna!"),end
df=divdif(X,Y);
n=length(df)-1;
YQ=df(n+1)*ones(size(XQ));
for i=n:-1:1 %algoritmo di horner
   YQ=YQ.*(XQ-X(i))+df(i);
end
return
end

function df=divdif(x,f)
%function per il calcolo delle differenze divise per il polinomio
%interpolante in forma di newton
n=size(x);
if(n~=length(f)), error("Dati errati!"), end
df=f;
n=n-1;
for i=1:n
   for j=n+1:-1:i+1
      df(j)=(df(j)-df(j-1))/(x(j)-x(j-i));
   end
end
return;
end
```

Per ricavarmi questa function è stato necessario in primo luogo ricavarmi il vettore delle differenze divise

grazie alla function divdif riportata in fondo che sfrutta la propietà delle differenze divise:

$$f[x_0, x_1, ..., x_{r-1}, x_r] = \frac{f[x_1, ..., x_{r-1}, x_r] - f[x_0, x_1, ...x_{r-1}]}{x_r - x_0}$$

Infine è stato sufficiente sfruttare l'algoritmo di horner per calcolare i valori che il polinomio assume nelle ascisse XQ e salvarli in YQ. Notare inoltre come tale function calcoli lo stesso valore della precedente function in quanto, pur ricavando i valori del polinomio interpolante in forme differenti, il polinomio di grado n interpolante una funzione in un insieme di n+1 ascisse è unico.

Esercizio 18
Costruire una function, hermite.m, avente una sintassi analoga alla function spline di Matlab (in pratica, con un parametro di ingresso in pi‘u per i valori delle derivate), che implementi, in modo vettoriale, il polinomio interpolante di Hermite.

Soluzione

```
function YQ = hermitte(X,Y, Y1, XQ)
%
% YQ=hermitte(X,Y,Y1,XQ)
%
%Input
%X: Ascisse d'interpolazione,
%Y: Valori che la funzione assume nelle ascisse d'interpolazione
%Y1: Valori che la derivata prima della funzione assume nelle ascisse
%d'interpolazione
%XQ: Vettore contenente le ascisse in cui vogliamo approssimare la funzione
%
%Output
%YQ: Valori approssimati della funzione con il polinomio interpolante di Hermitte in
%forma di Newton
%
%Calcola i valori approssimati della funzione(di cui conosciamo sia i valori Y che assume nelle
    ascisse X ed i valori Y1 la cui derivata prima assume nelle stesse ascisse)
%calcolati attraverso il polinomio interpolante in
%forma di Lagrange nelle ascisse XQ.
%
if(length(X)~=length(Y)), error("Dati in ingresso errati!"),end
if (length(X) ~= length(unique(X))), error("Ascisse d'interpolazione non distinte!"),end %uso la
    function unique che restuisce un vettore contenente i valori senza ripetizioni di X
if(length(Y1)~=length(Y)), error("Dati in ingresso errati!"),end
n=(length(X));
fi(1:2:2*n-1)=Y;
fi(2:2:2*n)=Y1;
df=diffdivhermitte(X,fi');
n=length(df)-1;
YQ=df(n+1)*ones(size(XQ)); %algoritmo di horner per il calcolo dei valori di un polinomio
for i=n:-1:1
    YQ=YQ.*(XQ-X(round(i/2,0)))+df(i);
end
return
end

function f=diffdivhermitte(x,f)
%function per calcolare le differenze divise per il polinomio interpolante
%di hermitte
n=(length(f)/2)-1;
for i=2*n+1:-2:3
    f(i)=(f(i)-f(i-2))/(x(round(i/2,0))-x(round((i-1)/2,0)));
end
for j= 2:2*n+1
    for i=(2*n+2):-1:j+1
        f(i)=(f(i)-f(i-1))/(x(round(i/2,0))-x(round((i-j)/2,0)));
    end
end
end
```

Questa function è molto simile a quella precedente se non per l'algoritmo delle differenze divise che, in questo

caso, include anche le derivate prime della funzione che intendiamo interpolare (oltre al fatto che, costruendo un polinomio di grado $2n+1$ dove $n+1$ è il numero di ascisse d'interpolazione, ho dovuto adattare l'algoritmo di horner di conseguenza).

Esercizio 19
Costruire una function Matlab che, specificato in ingresso il grado n del polinomio interpolante, e gli estremi dell'intervallo $[a, b]$, calcoli le corrispondenti ascisse di Chebyshev.

Soluzione

```
function x =cheby(n,a,b)
%x =cheby(n,a,b)
%Input
%n: grado del polinomio interpolante, a,b: estremi dell'intervallo
%di interpolazione
%Output
%x: ascisse di Chebyshev ricavate
%
%Ricava le ascisse di Chebyshev nell'intervallo [a,b]
%per un polinomio interpolante di grado n
if(n<0), error('Grado del polinomio interpolante non valido!'),end
if(a>=b), error('Intervallo definito in maniera non corretta!'),end
n=n+1;
x(n:-1:1)=(a+b)/2+((b-a)/2)*cos(((2*(1:n)-1)*pi)/(2*n));
```

In questo caso la function consiste di un solo passaggio fondamentale che non è altro che il calcolo ripetuto di

$$x_{n-i} = \frac{a+b}{2} + \frac{b-a}{2} cos\left(\frac{(2i+1)\pi}{2(n+1)}\right)$$

per i che va da 0 ad n grado del polinomio di cui vogliamo ricavarci le ascisse nell'intervallo $[a, b]$.

Esercizio 20
Costruire una function, spline0.m, avente la stessa sintassi della function spline di Matlab, che implementi la spline cubica naturale interpolante una funzione.

Soluzione

```
function YQ = spline0(X, Y, XQ)
% YQ=spline=(X,Y,XQ)
%
%Input
%X: Vettore colonna contenente le ascisse d'interpolazione,
%Y: Vettore colonna contenente i valori che la funzione assume nelle ascisse d'interpolazione
%XQ: Vettore contente le ascisse in cui vogliamo approssimare la funzione
%Output
%YQ: Valori approssimati della funzione interpolata attraverso una funzione
%spline cubica naturale
%
%Calcola i valori approssimati della funzione(di cui conosciamo i valori Y
%che assume nelle ascisse X) calcolati attraverso una funzione spline
%cubica naturale interpolante la funzione.
%
if(length(X)~=length(Y)), error("Lunghezza dei dati in ingresso non valida!"),end
if (length(X) ~= length(unique(X))), error("Ascisse d'interpolazione non distinte!"),end %uso la
    function unique che restuisce un vettore contenente i valori senza ripetizioni di X
if(isempty(XQ)), error("Il vettore contenente le ascisse in cui interpolare la funzione e'
    vuoto!"),end
if(size(X,2)>1||size(Y,2)>1),error("Inserire vettori colonna!"),end
n=length(X)-1;
eps=zeros(n-1,1);
p=zeros(n-1,1);
for i = 1 : n - 1
   p(i)=(X(i+1)-X(i))/(X(i+2)-X(i));
   eps(i)=(X(i+2)-X(i+1))/(X(i+2)-X(i));
end
dif = diffdivspline(X, Y);
m = tridia(2*ones(n-1,1), p, eps,dif*6);
m=[0;m;0];
YQ = calcolapuntixq(X, Y, m, XQ);
end

function[ df ] = diffdivspline(x, f)
%calcola le differenze divise fermandosi pero' alla seconda iterazione
% per ottenere le differenze divise tra tre punti che ci serviranno per calcolare m
n=size(x);
df=f;
n=n-1;
for j=1:2
   for i=n+1:-1:j+1
      df(i)=(df(i)-df(i-1))/(x(i)-x(i-j));
   end
end
df=df(3:n+1);
return;
end

function x=tridia(a,b,c,x)
%calcola il sistema tridiagonale in cui, nel nostro caso, il vettore a
%contiene solo 2 mentre i vettori b e c rispettivamente i valori p ed eps
%mentre x e' il vettore delle differenze divise per ricavare il vettore
```

```
%contente i valori m
n=length(a);
for i=1:n-1                  % mi ricavo una fattorizzazione LU della matrice tridiagonale
    risolvendo contemporaneamente il fattore L
   b(i)=b(i)/a(i);
   a(i+1)=a(i+1)-b(i)*c(i);
   x(i+1)=x(i+1)-b(i)*x(i);
end
x(n)=x(n)/a(n);
for i=n-1:-1:1
   x(i)=(x(i)-c(i)*x(i+1))/a(i); %risolvo il fattore U rimanente della scomposizione LU
end
return
end

function s = calcolapuntixq( xi, fi, m, XQ)
%Calcola i valori della spline precedentemente ricavata grazie ai valori m
%nei punti XQ anche se essi si trovano al di fuori dell'intervallo d'interpolazione
n = length(xi)-1;
z=length(XQ);
s=zeros(z,1);
[xi,p]=sort(xi); %ordino i vettori delle ascisse in modo che formino una partizione vera e propria
    e di conseguenza ordino anche il vettore dei corrispondenti valori della funzione in tali
    ascisse
fi=fi(p);
for j=1:z
    for i=2:n+1
        if((XQ(j)>=xi(i-1)&& XQ(j)<=xi(i))||XQ(j)<xi(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));
            s(j) =((XQ(j)-xi(i-1))^3*m(i)+(xi(i)-XQ(j))^3*m(i-1))/(6*hi)+qi*(XQ(j)-xi(i-1))+ri;
            break
        elseif(XQ(j)>xi(n+1))   %questa parte serve solo per fare in modo che, se anche cerco di
            approssimare un valore della funzione in un ascissa piu' grande dell'estremo superiore
            dell'intervallo d'interpolazione,
            % la function restituisca comunque un valore che e' calcolato nella funzione
                dell'ultimo tratto da cui e' composta la spline (come avviene per la function
                spline)
            hi=xi(n+1)-xi(n);
            ri=fi(n)-hi^2/6*m(n);
            qi=(fi(n+1)-fi(n))/hi-hi/6*(m(n+1)-m(n));
            s(j) =((XQ(j)-xi(n))^3*m(n+1)+(xi(n+1)-XQ(j))^3*m(n))/(6*hi)+qi*(XQ(j)-xi(n))+ri;
            break
        end
    end
end
end
```

In questa function abbiamo dovuto tenere di conto il fatto che la function deve comunque restituire un valore anche se cerchiamo di interpolare la funzione in un punto esterno all'intervallo d'interpolazione (in particolare per l'esercizio 21).

Esercizio 21
Costruire una tabella in cui viene riportato, al crescere di n, il massimo errore di interpolazione ottenuto approssimando la funzione:

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

sulle ascisse x0 < x1 · · · < xn:
• equidistanti in [-2, 3],
• di Chebyshev per lo stesso intervallo,
utilizzando le function degli Esercizi 16–18 e 20, e la function spline di Matlab. Considerare n = 4, 8, 16, . . . , 40 e stimare l'errore di interpolazione su 10001 punti equidistanti nell'intervallo [x0, xn].

Soluzione
Di seguito riportiamo la tabella con il massimo errore per ciascuna delle function viste in precedenza per polinomi di vario grado con n+1 ascisse equidistanti:

n	lagrange	newton	hermitte	spline0	spline
4	0.4384	0.4384	0.2236	0.2793	0.3171
8	1.0452	1.0452	1.1440	0.0562	0.0561
12	3.6634	3.6634	14.0120	0.0069	0.0069
16	14.3939	14.3939	216.0456	0.0037	0.0037
20	59.8223	59.8223	3729.3	0.0032	0.0032
24	257.2129	257.2129	68913.9	0.0019	0.0019
28	1131.42	1131.42	1.33e+06	0.0010	0.0010
32	5058.96	5058.96	2.66e+07	6.55e-04	6.55e-04
36	2.29e+04	2.29e+04	5.72e+08	4.21e-04	4.21e-04
40	1.05e+05	1.05e+05	2.37e+10	2.77e-04	2.77e-04

Nella prossima invece mostriamo l'errore massimo delle stesse function calcolate però nelle ascisse di Chebyshev:

n	lagrange	newton	hermitte	spline0	spline
4	0.4020	0.4020	0.2672	0.3316	0.3592
8	0.1708	0.1708	0.0725	0.1354	0.1339
12	0.0692	0.0692	0.0174	0.0479	0.0471
16	0.0326	0.0326	0.0039	0.0.0140	0.0136
20	0.0153	0.0153	8.3913-04	0.0022	0.0020
24	0.0069	0.0069	0.0019	0.0036	0.0035
28	0.0031	0.0031	0.184	0.0035	0.0035
32	0.0014	0.0014	30.395	0.0028	0.0028
36	6.4075e-04	6.4075e-04	6.14e+03	0.0020	0.0020
40	2.8946e-04	2.8946e-04	4.55e+05	0.0014	0.0014

Una crescita tale dell'errore nelle function della prima tabella (tranne le spline) è spiegata dal problema del condizionamento nell'interpolazione che è dovuto alla costante di Lebesgue (che tende all'infinito all'aumentare del grado del polinomio interpolante e quindi delle ascisse di interpolazione) che è il numero di condizionamento di tale problema che ha crescita esponenziale nel caso di ascisse equidistanti mentre ha crescita pari a $\frac{2}{\pi} logn$ nel caso si scelgano le ascisse di Chebyshev. Inoltre va ricordato che il grado del polinomio interpolante di Hermitte è $2n + 1$ sulle stesse ascisse di un polinomio interpolante in forma di Lagrange o Newton di grado n (quindi su $n + 1$ ascisse d'interpolazione). Possiamo anche notare come il polinomio in forma di Lagrange e quello in forma di Newton abbiano lo stesso errore massimo in quanto sono a tutti gli effetti lo stesso polinmomio scritto in forma differente.
Osserviamo inoltre come l'errore massimo, nelle function spline e spline0, sia maggiore usando le ascisse di Chebyshev rispetto a quelle equidistanti in quanto una spline cubica interpolante non risente del problema del condizionamento delle funzioni interpolanti precedentemente descritto. Infine va notato che è stato necessario ricavare la derivata della funzione,che riportiamo di seguito, per poter ricavare il polinomio di

Hermitte:

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

Esercizio 22
Tabulare il massimo errore di approssimazione (stimato su 10001 punti equidistanti in [0, 1]) ottenuto approssimando le funzioni
$sin(2\pi x)$ e $cos(2\pi x)$
mediante le function spline0 e spline, interpolanti su n + 1 punti equidistanti in [0, 1], per $n = 5, 10, 15, 20, ..., 50$. Commentare i risultati ottenuti.

Soluzione

Di seguito riportiamo la tabella (rispettivamente per seno e coseno) con il massimo errore per ciascuna delle due funzioni al crescere di n (numero dei punti meno uno):

Funzione $sin(2\pi x)$

Grado	spline0	spline
5	0.0090	0.0648
10	0.00045	0.0026
15	8.329e-05	3.635e-04
20	2.567e-05	8.751e-05
25	1.053e-05	2.887e-05
30	5,066e-06	1.164e-05
35	2.724e-06	5.398e-06
40	1.590e-06	2.772e-06
45	9.940e-07	1.540e-06
50	6.520e-07	9.099e-07

Funzione $cos(2\pi x)$

Grado	spline0	spline
5	0.0951	0.0181
10	0.0204	0.0034
15	0.0088	7.878e-04
20	0.0049	2.605e-04
25	0.0031	1.088e-04
30	0.0022	5.307e-05
35	0.0016	2.883e-05
40	0.0012	1.697e-05
45	9.595e-04	1.0626e-05
50	7.768e-04	6.986e-06

Possiamo notare come la function spline0 interpoli la funzione $sin(2\pi x)$ con un errore molto minore rispetto alla funzione $cos(2\pi x)$, questa differenza è dovuta al fatto che la function spline0 implementa una spline cubica interpolante naturale, cioè una spline cubica interpolante a cui abbiamo imposto la condizione che negli estremi dell'intervallo d'interpolazione il valore della derivata seconda sia uguale a zero, cosa che nella funzione $sin(2\pi x)$ è vera nel nostro intervallo $[0, 1]$ in quanto la derivata seconda di tale funzione risulta essere $-4\pi^2 sin(2\pi x)$ che, calcolate in 0 ed 1, risultano essere uguali a 0. L'altra funzione invece non possiede questa caratteristica e per questo la precisione dell'approssimazione sarà ridotta rispetto alla funzione precedentemente citata.
La function spline non posssiede invece tale caratteristica in quanto implementa una spline cubica interpolante not-a-knot ossia nella quale abbiamo impostp che il primo tratto di funzione, con estremi le ascisse d'interpolazione x_0 e x_1, è uguale al secondo che ha come estremi x_1 e x_2 e allo stesso modo il penultimo dovrà uguale all'ultimo tratto, motivo per il quale non c'è grande differenza fra gli errori delle due funzioni interpolate con la funzione spline.

Esercizio 23

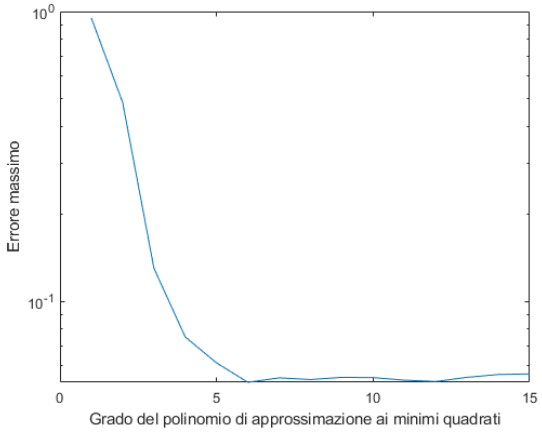
Sia assegnata la seguente perturbazione della funzione $f(x) = sin(\pi x^2)$):
$\tilde{f}(x) = f(x) + 10^{-1} rand(size(x))$,
in cui rand 'e la function built-in di Matlab. Calcolare polinomio di approssimazione ai minimi quadrati di grado m, p(x), sui dati $(xi, \tilde{f}(xi)), i = 0, ..., n$, con: $xi = i/n, n = 10^4$

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

Soluzione
Di seguito è riportato lo script usato per trovare il vettore contenente il massimo errorre per ciascuno dei vari polinomi di grado da 1 a 15

```
f=@(x) sin(pi.*x.^2); %funzione non perturbata
fper=@(x) f(x)+0.1*rand(size(x)); %funzione perturbata
n=10^4;
x=(1:n)./n;
yper=zeros(size(x));
for i=1:n
    yper(i)=fper(x(i));
end
van=fliplr(vander(x)); %creo la matrice di vandermonde
coef=zeros(15,15);
errmax=zeros(15,1);
for m=1:15
    coef(1:m+1,m)=miaqr(van(:,1:m+1),yper'); %ottengo il vetttore dei coefficienti dei vari
        polinomi di approssimazione ai minimi quadrati
end
p=zeros(size(x,2),15);
error=zeros(size(p));
y=f(x);
for m=1:15
p(:,m)=horner(coef(:,m),x')'; %sfrutto l'algoritmo di Horner per calcolare i valori dei vari
    polinomi nei punti che mi interessano
error(1:n,m)=abs(p(1:n,m)-y');
errmax(m)=max(error(1:n,m)); %mi ricavo il massimo valore dell'errore per ciascuno dei polinomi di
    vario grado
end
semilogy(1:15,errmax); %disegno il grafico semilogy dell'errore massimo in base al grado m del
    polinomio
```

Questo script ha dato come risultato il seguente grafico:



Possiamo ossevare come sia inutile incrementare oltre 6 il grado del polinomio di approssimazione ai minimi quadrati in quanto, non solo l'errore non diminuisce all'aumentare del grado del polinomio, ma anzi sembrerebbe crescere all'aumentare del grado. Infine riportiamo anche la function per calcolare i valori di un polinomio sfruttando l'algoritmo di horner;

```
function p=horner(a,x)
%p=horner(a,x)
%
%calcola il polinomio con coefficienti a nei valori x
n=length(a);
p=ones(length(x),1).*a(n);
for k=n-1:-1:1
    p=p.*x+a(k);
end
```

Esercizio 24
Costruire una function Matlab che, dato in input n, restituisca i pesi della quadratura della formula di Newton-Cotes di grado n. Tabulare, quindi, i pesi delle formule di grado 1, 2,. . . , 7 e 9 (come numeri razionali).

Soluzione

```
function coef=calcolacoefficientigrado(n)
%
% coef=calcolacoefficientigrado(n)
%
% Input
%n: Grado (maggiore di 0) della formula di Newton-Cotes di cui vogliamo conoscere i pesi
%della quadratura.
%
%Output
%coef: pesi della quadratura della formula di grado desiderato
%
%Calcola i pesi della quadratura della formula di quadratura di
%Newton-Cotes di grado n.
if(n<=0), error("Valore del grado della formula di Newton-Cotes non valido"),end
coef=zeros(n+1,1);
if (mod(n,2) == 0)
    for i=0:n/2-1
        coef(i+1)=calcolacoefficienti(i,n);
    end
    coef(n/2+1)=n-sum(coef)*2;
    coef((n/2)+1:n+1)=coef((n/2)+1:-1:1);
else
    for i=0:round(n/2,0)-2
        coef(i+1)=calcolacoefficienti(i,n);
    end
    coef(round(n/2,0))=(n-sum(coef)*2)/2;
    coef(round(n/2,0)+1:n+1)=coef(round(n/2,0):-1:1);
end
return
end

function cin=calcolacoefficienti(i,n)
%calcola il peso della quadratura della formula di Newton-Cotes numero i di
%grado n
d=i-[0:i-1 i+1:n];
den=prod(d);
a=poly([0:i-1 i+1:n]);
a=[a./((n+1):-1:1) 0];
num=polyval(a,n);
cin=num/den;
end
```

Possiamo notare come, essendo i pesi di quadratura simmetrici, è stato sufficiente calcolare meno della metà dei pesi richiesti considerando anche che la somma di tutti i pesi di quadratura è uguale al grado n della formula di Newton-Cotes di cui stiamo calcolando i pesi di quadratura.

I pesi della quadratura ricavati con la precedente function al variare del grado n sono riportati nei vettori sottostanti:

$$n = 1$$

$$\begin{bmatrix} \frac{1}{2} \\ \\ \frac{1}{2} \end{bmatrix}$$

$$n = 2$$

$$\begin{bmatrix} \frac{1}{3} \\ \\ \frac{4}{3} \\ \\ \frac{1}{3} \end{bmatrix}$$

$$n = 3$$

$$\begin{bmatrix} \frac{3}{8} \\ \\ \frac{9}{8} \\ \\ \frac{9}{8} \\ \\ \frac{3}{8} \end{bmatrix}$$

$$n = 4$$

$$\begin{bmatrix} \frac{14}{45} \\ \\ \frac{64}{45} \\ \\ \frac{8}{15} \\ \\ \frac{64}{45} \\ \\ \frac{14}{45} \end{bmatrix}$$

$$n = 5$$

$$\begin{bmatrix} \frac{95}{288} \\ \\ \frac{125}{96} \\ \\ \frac{125}{144} \\ \\ \frac{125}{144} \\ \\ \frac{125}{96} \\ \\ \frac{95}{288} \end{bmatrix}$$

$n = 6$

$$\begin{bmatrix} \frac{41}{140} \\ \frac{54}{35} \\ \frac{27}{140} \\ \frac{68}{35} \\ \frac{27}{140} \\ \frac{54}{35} \\ \frac{41}{140} \end{bmatrix}$$

$n = 7$

$$\begin{bmatrix} \frac{1073}{3527} \\ \frac{810}{559} \\ \frac{343}{640} \\ \frac{649}{536} \\ \frac{649}{536} \\ \frac{343}{640} \\ \frac{810}{559} \\ \frac{1073}{3527} \end{bmatrix}$$

$n = 9$

$$\begin{bmatrix} \frac{130}{453} \\ \frac{1374}{869} \\ \frac{243}{2240} \\ \frac{5287}{2721} \\ \frac{704}{1213} \\ \frac{704}{1213} \\ \frac{5287}{2721} \\ \frac{243}{2240} \\ \frac{1374}{869} \\ \frac{130}{453} \end{bmatrix}$$

Esercizio 25
Utilizzare le formule tabulate nel precedente esercizio per calcolare le approssima- zioni dell'integrale

$$\int_0^1 e^{3x} dx,$$

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

Soluzione

Anzitutto è stato necessario calcolare a mano l'integrale definito sopra riportato. Abbiamo quindi calcolato che:

$$\int_0^1 e^{3x} dx = \frac{1}{3}e^3 - \frac{1}{3}$$

Successivamente abbiamo calcolato la soluzione con la function sotto riportata:

```
function int =integral(a,b,f,n)
%
% int =integral(a,b,f,n)
%
% Input
%
%f: Function handler contenente la funzione di cui vogliamo calcolare l'approssimazione
    dell'integrale definito,
%a,b: Estremi dell'intervallo d'integrazione,
%n: Grado della formula di Newton-Cotes desiderato,
%Output
%int: approssimazione dell'integrale ottenuta
%
%Calcola un'approssimazione dell'integrale definito di f tra a e b con la
%formula di Newton-Cotes di grado n
%Newton-Cotes di grado n.
if(n<=0), error('Grado della formula di Newton-Cotes non valido!'),end
x=linspace(a,b,n+1)';
y=f(x);
int=(b-a)/n*sum(y.*calcolacoefficientigrado(n));
return;
end
```

Ottenendo i risultati che abbiamo tabulato, insieme all'errore, nella tabella qui sotto:

Grado	risultato	errore
1	10.5428	4.1809
2	6.5020	0.1402
3	6.4259	0.0641
4	6.3637	0.0018
5	6.3629	0.0010
6	5,6.3619	1.9901e-05
7	6.3619	1.2261e-05
9	6.3618	1.0500e-07

Esercizio 26
Scrivere una function Matlab,
[If,err,nfeval] = composita(fun, a, b, n, tol)
in cui:
- fun è l'identificatore di una function che calcoli (in modo vettoriale) la funzione integranda,
- a e b sono gli estremi dell'intervallo di integrazione,
- n è il grado di una formula di Newton-Cotes base,
- tol è l'accuratezza richiesta,

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

Soluzione

```
function [If,err,nfeval] = composita(fun,a,b,n,tol)
% [If,err,nfeval] = composita( fun, a, b, n, tol)
%
%Input
%fun: Function handler contenente la funzione di cui vogliamo calcolare l'approssimazione
    dell'integrale definito,
%a,b: Estremi dell'intervallo d'integrazione,
%n: Grado della formula di Newton-Cotes desiderato,
%tol: Tolleranza richiesta
%Output
%If: approssimazione dell'integrale ottenuta,
%err: stima dell'errore dell'approssimazione,
%nfeval: numero di valutazioni funzionali effettuate,
%
%Calcola un'approssimazione dell'integrale definito della funzione fun
%con estremi a e b utilizzando la formula composita di Newton-Cotes
%di grado n con tollerenza tol
%
if(n<=0), error('Grado della formula di Newton-Cotes non valido!'),end
if(tol<=0), error('Tolleranza non valida!'),end
x=linspace(a,b,n+1)';
coef=calcolacoefficientigrado(n); %function vista nell'esercizio 24
y=fun(x);
nfeval=n+1;
u=1;
if (mod(n,2) == 0)
    u=2;
end

int=integrale(a,b,y,n,coef);
dn=2*(n+1);
int2=0;
for i=1:1000
       xi=linspace(a,b,dn-1)';
       y2=zeros(dn-1,1);
       y2(1:2:dn)=y;
       y2(2:2:dn-1)=(fun(xi((2:2:dn-1))));
       nfeval=nfeval+(dn/2-1); %aggiorno le valutazioni funzionali richieste
       inf=1;sup=n+1;
       for j=1:1:2^i
           int2=int2+integrale((j-1)*(b-a)/(2^i),(j)*(b-a)/(2^i),y2(inf:sup),n,coef); %sommo tutti
               i sottointervalli dell'integrale approssimato con le formule di Newton-Cotes
```

```
            inf=sup;
            sup=sup+n;
        end
        err=(int2-int)/(2^(n+u)-1); %calcolo una stima dell'errore
        if(abs(err)<tol), If=int2;break,end %se la stima rispetta la tolleranza interrompo il ciclo
        y=y2;
        dn=size(y,1)*2;
        int=int2;
        int2=0;
end
return
end

function int =integrale(a,b,y,n,coef)
%Calcola la formula di Newton-Cotes tra a e b di grado n con coefficienti
%coef e con y contenente i valori che la funzione assume nelle n+1 ascisse equistanti
    nell'intervallo [a,b]
int=(b-a)/n*sum(y.*coef);
return;
end
```

Per realizzare questa function è stata usata la formula:

$$E_k^{(n)} \approx \frac{I_k^{(n)} - I_k^{(\frac{n}{2})}}{2^{k+u} - 1}$$

per stimare l'errore (ossia la differenza tra il valore dell'integrale definito della funzione tra nell'intervallo con estremi a e b e la nostra approssimazione) in cui il valore u vale 1 se k è dispari mentre vale 2 se u è pari. Tale stima quindi non ci garantisce di conoscere il valore reale dell'errore il che potrebbe, soprattuto nel caso di funzioni non sufficientemnte regolari, portare a stimare incorrettamentre l'errore e quindi anche ad approssimare in maniera erratta l'integrale stesso.

Esercizio 27
Tabulare il numero di valutazioni di funzione richieste per calcolare, mediante la function del precedente esercizio, l'approssimazione dell'integrale

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

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

Soluzione
Tabella con n=1:

nfeval	tolleranza
5	1e-02
129	1e-03
513	1e-04
1025	1e-05
4097	1e-06

Tabella con n=2:

nfeval	tolleranza
5	1e-02
65	1e-03
129	1e-04
257	1e-05
513	1e-06

Tabella con n=3:

nfeval	tolleranza
13	1e-02
13	1e-03
97	1e-04
385	1e-05
385	1e-06

Tabella con n=4:

nfeval	tolleranza
9	1e-02
33	1e-03
65	1e-04
129	1e-05
257	1e-06

Tabella con n=5:

nfeval	tolleranza
11	1e-02
11	1e-03
81	1e-04
161	1e-05
321	1e-06

Tabella con n=6:

nfeval	tolleranza
13	1e-02
13	1e-03
49	1e-04
95	1e-05
193	1e-06

Tabella con n=7:

nfeval	tolleranza
15	1e-02
14	1e-03
57	1e-04
113	1e-05
225	1e-06

Tabella con n=9:

nfeval	tolleranza
19	1e-02
19	1e-03
37	1e-04
37	1e-05
37	1e-06

Esercizio 28
Scrivere una function che implementi la formula adattiva dei trapezi. N.B.: evitare di effettuare valutazioni di funzione ridondanti.

Soluzione

```
function [I2,nfeval] = adattivatrap( f, a, b, tol, fa, fb )
% [I2,nfeval] = adattivatrap( f, a, b, tol)
%
%Input
%f: Function handler contenente la funzione di cui vogliamo calcolare l'approssimazione
    dell'integrale definito,
%a,b: Estremi dell'intervallo d'integrazione,
%tol: Tolleranza richiesta
%(fa,fb): parametri di lavoro da non specificare quando si vuole invocare
%la funzione
%
%Output
%If: approssimazione dell'integrale ottenuta,
%nfeval: numero di valutazioni funzionali effettuate,
%
%Calcola un'approssimazione dell'integrale definito della funzione fun
%con estremi a e b utilizzando la formula adattiva di Newton-Cotes
%di grado 1 con tollerenza tol
%
if(tol<=0), error('Tolleranza non valida!');end
nfeval=0;
if nargin<=4
fa=f(a);
fb=f(b);
nfeval=2;
end
nfeval=nfeval+1; %aggiorno il numero di valutazioni funzionali richieste
h=b-a;
xm=(a+b)/2; %calcolo il punto medio
fm=f(xm);
I1=(h/2)*(fa+fb); %Applico la formula dei trapezzi
I2=(I1+h*fm)/2; %Applico la formula composita dei trapezzi su fa,fm e fb
er=abs(I2-I1)/3; %mi ricavo una stima dell'errore
if er>tol %se la tolleranza non e' rispettata riapplico la function sui due sottointervalli
    [lI,nfevall]=adattivatrap( f,a, xm, tol/2, fa, fm );
    [rI,nfevalr]=adattivatrap(f,xm, b, tol/2, fm, fb );
    I2=lI+rI;
    nfeval=nfeval+nfevalr+nfevall; %sommo le valutazioni funzionali richieste nei due
        sottointervalli
end
return
end
```

Notare come, per evitare valutazioni funzionali ridondanti, i valori della funzione di cui approssimare l'integrale negli estremi dei due sottointervalli da integrare (che verrano in seguito sommati) vengano passati come argomenti evitando quindi di doverli calcolare nuovamente. Infine per stimare l'errore è stata usata la stessa formula vista per la function composita (con la maggior parte dei valori già calolata in quanto useremo solo la formula dei trapezzi).

Esercizio 29
Scrivere una function che implementi la formula adattiva di Simspon. N.B.: evitare di effettuare valutazioni di funzione ridondanti.

Soluzione

```
function [I2,nfeval] = adattivasimp( f, a, b, tol, fa, fm,fb )
% [I2,nfeval] = adattivasimp( f, a, b, tol)
%
%Input
%f: Function handler contenente la funzione di cui vogliamo calcolare l'approssimazione
    dell'integrale definito,
%a,b: Estremi dell'intervallo d'integrazione,
%tol: Tolleranza richiesta
%(fa,fm,fb): parametri di lavoro da non specificare quando si vuole invocare
%la funzione
%
%Output
%If: approssimazione dell'integrale ottenuta,
%nfeval: numero di valutazioni funzionali effettuate,
%
%Calcola un'approssimazione dell'integrale definito della funzione f
%con estremi a e b utilizzando la formula adattiva di Newton-Cotes
%di grado 2 con tollerenza tol
%
if(tol<=0), error('Tolleranza non valida!'),end
m = ( a + b )/2; %calcolo il punto medio
nfeval=0;
if nargin<=4
fa = f(a);
fb = f(b);
fm = f(m);
nfeval=3;
end
nfeval=nfeval+2; %aggiorno il numero di valutazioni funzionali richieste
ma=(a+m)/2; %calcolo il punto medio dell'intervallo [a,m]
mb=(b+m)/2; %calcolo il punto medio dell'intervallo [m,b]
fma=f(ma);
fmb=f(mb);
h = b - a;
I1= ( h/6 )*( fa +4*fm+ fb); %Applico la formula di Simpson su fa,fm ed fb
I2 = I1/2 + (h/6)*(2*fma + 2*fmb-fm); %Applico la formula composita di Simpson su fa,fma,fm,fmb e
    fb
e = abs( I2 - I1 )/15; %mi ricavo una stima dell'errore
if e>tol %se la tolleranza non e' rispettata riapplico la function sui due sottointervalli
    [lI,nfevall]=adattivasimp(f,a, m, tol/2, fa, fma, fm );
    [rI,nfevalr]=adattivasimp(f,m, b, tol/2, fm, fmb, fb );
    I2=lI+rI;
    nfeval=nfeval+nfevalr+nfevall; %sommo le valutazioni funzionali richieste nei due
        sottointervalli
end
return
end
```

Anche in questo caso possiamo notare come si siano evitate valutazioni funzionali ridondanti passando come argomento alla stessa function, applicata sui due sottointervalli, oltre ai valori che la function assume negli estremi anche i valori che essi assumono nei loro punti medi (che erano stati calcolati in precedenza per calcolare la formula di Simpson composita per approssimare l'integrale "raddoppiando" i punti per poter ottenere una stima dell'errore). Infine anche in questo per stimare l'errore è stata usata la stessa formula vista per la function composita (con la maggior parte dei valori già calolata in quanto useremo solo la formula di Simpson).

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

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

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

Soluzione

Di seguito riportiamo le tabelle con il numero di valutazioni funzionali per raggiungere una stima dell'errore congrua con l'errore desiderato:

nfeval trapezzi	nfeval Simpson	tolleranza
5	5	1e-02
101	25	1e-03
265	49	1e-04
971	69	1e-05
3151	129	1e-06

Si può notare come la non regolarità della funzione di cui volgiamo approssiamre l'integrale richieda alla formula adattiva dei trapezi un numero molto più cospicuo (con una crescita molto veloce al diminuire della soglia di tolleranza) di valutazioni funzionali rispetto alla formula adattiva di Simpson che richiede invece un numero molto inferiore di valutazioni funzionali soprattuto con le soglie di tolleranza più inferiori. Infine notiamo come tali formule richiedano meno valutazioni funzionali rispetto alle loro corrispettive composite (come possiamo vedere nell'esercizio 27 in cui approssimiamo lo stesso integrale con le formule di Newton-Cotes composite). Tale differenza era prevedibile in quanto le formule di Newton-Cotes adattive sono pensate per evitare di dover effettuare valutazioni funzionali inutili.