

Elaborato di Calcolo Numerico

Finoia Luca 6132811 Calabrese Filippo 5826217

July 8, 2019

CHAPTER 1

ERRORI ED ARITMETICA FINITA

1.1 Esercizio 1

Descrizione: Verificare che, per h sufficientemente piccolo, $\frac{3}{2}f(x) - 2f(x - h) + \frac{1}{2}f(x - 2h) = hf'(x) + O(h^3)$.

Soluzione:

considerando il Polinomio di Taylor centrato in x_0 al secondo ordine:

$$f(x_0 - h) = f(x_0) - f'(x_0)h + \frac{1}{2}f''(x_0)h^2 + O(h^3) \quad (1.1)$$

si effettua una sostituzione con $f(x_0 - h)$ nella funzione e si ottiene:

$$\begin{aligned} & \frac{2}{3}f(x) - 2 \left[f(x) - f'(x)h + \frac{1}{2}f''(x)h^2 + O(h^3) \right] + \\ & + \frac{1}{2} \left[f(x) - f'(x)2h + \frac{1}{2}f''(x)4h^2 + O(h^3) \right] = \\ & = hf'(x) + O(h^3) \end{aligned} \quad (1.2)$$

da cui otteniamo:

$$\begin{aligned} & \frac{3}{2}f(x) - 2f(x) + 2f'(x)h - \\ & - f''(x)h^2 + \frac{1}{2}f''(x)h^2 - \\ & - f'(x)h + f''(x)h^2 + O(h^3) = \\ & = hf'(x) + O(h^3) \end{aligned} \quad (1.3)$$

quindi per h sufficientemente piccoli si ottiene:

$$hf'(x) + O(h^3) = hf'(x) + O(h^3) \quad (1.4)$$

1.2 Esercizio 2

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

Soluzione:

L'insieme dei numeri di macchina \mathcal{M} è un insieme finito di elementi, tramite i quali è possibile rappresentare un insieme denso \mathcal{I} e che rappresenta un sottoinsieme dei numeri reali $\mathcal{I} \subset \mathbb{R}$.

$$\mathcal{I} = [-real_{max}, -real_{min}] \cup \{0\} \cup [real_{min}, real_{max}] \quad (1.5)$$

con $real_{max}$ e $real_{min}$ che indicano rispettivamente il più grande ed il più piccolo in valore assoluto tra i numeri macchina diversi da 0.

Per trovare i numeri di macchina normalizzati che possono essere ottenuti della doppia precisione IEEE possiamo considerare la seguente formula

$$b^s \cdot b^m \cdot (b^e - b^s) \quad (1.6)$$

dove indichiamo con b la base utilizzata, s i bit riservati per il segno, m i bit riservati per la mantissa ed e i bit riservati per l'esponente.

Considerando che lo standard *ANSI/IEEE 754-1985* utilizza una base binaria, per la rappresentazione dei numeri reali in singola e doppia precisione, abbiamo $b = 2$ indipendentemente dalla precisione utilizzata. Inoltre, per il formato a doppia precisione abbiamo 1 bit per il segno ($s = 1$), 52 bit per la mantissa ($m = 52$) e 11 bit per l'esponente ($e = 11$),

andando a sostituire i valori appena ricavati nella formula (1.6) otteniamo i numeri di macchina normalizzati che possono essere ottenuti della doppia precisione IEEE, quindi:

$$\begin{aligned} &= 2^1 \cdot 2^{52} \cdot (2^{11} - 2^1) = \\ &= 2^{1+52} \cdot (2^{11} - 2^1) = \\ &= 2^{1+52+11} - 2^{1+52+1} = \\ &= 2^{64} - 2^{54} = 18,428,729,675,200,069,632 \end{aligned} \quad (1.7)$$

1.3 Esercizio 3

Descrizione: Eseguire il seguente script Matlab e spiegare i risultati ottenuti:

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

Svolgimento:

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

u =

    1.0000000000000000e-300

>> u=1e-300; for i=1:n, u=u/2; end, for i=1:n, u=u*2; end, u

u =

    1.119916342203863e-300

>> |
```

in questo script Matlab viene moltiplicato per 2 un valore $u=1e-300$ per 75 volte, successivamente questo valore u viene diviso per 2 per 75 volte e infine si visualizza il valore in u , poi, viene riportato u al valore di partenza, cioè $u=1e-300$, e viene prima diviso per 2 per 75 volte, poi moltiplicato per 2 per 75 volte e infine si visualizza il valore in u .

Se fossimo in matematica esatta, in entrambi i casi, avremmo u che sarebbe uguale al valore di partenza poiché è stato prima moltiplicato e diviso per la stessa cifra (2^{75}) e poi diviso e moltiplicato per la stessa cifra (2^{75}). Tuttavia in Matlab i calcoli vengono effettuati in matematica finita e pertanto nel primo caso, in cui la moltiplicazione precede la divisione otteniamo lo stesso risultato che otterremmo in matematica esatta ovvero $1.0000000000000000e-300$, nel secondo caso invece otteniamo un valore differente ovvero $1.119916342203863e-300$.

La causa per cui nel secondo caso otteniamo un valore diverso da quello iniziale è da attribuire ad un errore di round-off, per la precisione un errore di underflow, causato nel primo ciclo in cui viene u viene diviso per 2 per 75 volte, dove, durante una delle iterazioni viene a verificarsi che, $0 < |u| < r_{min}$. Essendo

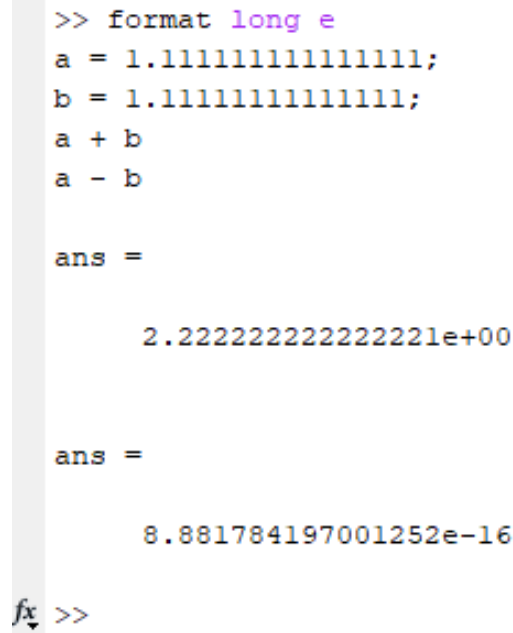
che r_{min} é il piú piccolo numero di macchina diverso da zero rappresentabile in valore assoluto, il valore di u viene arrotondato a questo, generando cosí un errore di underflow .

1.4 Esercizio 4

Descrizione: Eseguire le seguenti istruzioni Matlab e spiegare i risultati ottenuti:

```
format long e
a = 1.1111111111111111
b = 1.1111111111111111
a + b
a - b
```

Soluzione:



```
>> format long e
a = 1.1111111111111111;
b = 1.1111111111111111;
a + b
a - b

ans =

    2.2222222222222221e+00

ans =

    8.881784197001252e-16

fx >>
```

In questo script matlab vengono sommati due numeri a e b e poi viene fatta la differenza tra i due numeri a e b .

Nel caso della somma tra i due numeri il risultato ottenuto in Matlab corrisponde con il risultato ottenuto in matematica esatta in quanto una somma di numeri concorde é sempre ben condizionata poiché abbiamo

$$|\varepsilon_y| \leq \frac{|a| + |b|}{|a + b|} \varepsilon_x \equiv k \cdot \varepsilon_x \quad (1.8)$$

in cui k é il numero di condizionamento

$$k = \frac{|a| + |b|}{|a + b|} \quad (1.9)$$

e per una somma di numeri concorde $k=1$ e quindi é sempre ben condizionata.

Nel caso della differenza tra i due numeri a e b invece il risultato ottenuto in Matlab differisce dal risultato ottenuto in matematica esatta perché esso é 0.000000000000001 . In questo caso si verifica quindi il fenomeno della cancellazione numerica in cui si ha una perdita di cifre significative nel momento in cui viene effettuata una somma tra due numeri prossimi in valore assoluto ma opposti in segno. Infatti in questo caso, in cui $a \approx b$, si ottiene un numero di condizionamento arbitrariamente grande e quindi l'operazione di somma algebrica risulta mal condizionata.

CHAPTER 2

RADICI DI UN'EQUAZIONE

2.1 Esercizio 5

Descrizione: 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, utilizzando come criterio di arresto $|\Delta x_i| \leq \text{tol} \cdot (1 + |x_i|)$ essendo tol una opportuna tolleranza specificata in ingresso.

Soluzione:

il metodo di bisezione:

```
bisezione.m  X  +
1  function [x,i] = bisezione(fun,a,b,tolx)
2  %   x=bisezione(fun,a,b,tolx)
3  %   parametri in ingresso
4  %   fun -> funzione in cui si effettua la ricerca degli zeri
5  %   a,b -> estremi dell'intervallo di ricerca
6  %   tolx -> parametro che indica la tolleranza su x
7  %   valori di ritorno
8  %   x -> radice della funzione
9  %   i -> numero di iterazioni
10 %   il metodo di bisezione restituisce la radice di una funzione fun in
11 %   un intervallo [a,b] ed il numero di iterazioni effettuate
12 - if a>b, error('intervallo non valido'),end
13 - fa=feval(fun,a);
14 - fb=feval(fun,b);
15 - if fa*fb>0,error('nessuna radice in [a,b]'),end
16 - if fa==0,x=a;return; elseif fb==0,x=b;return, end
17 - nmax = ceil (log2((b-a)/tolx));
18 - for i=1:nmax
19 -     df=abs((fb-fa)/(b-a));
20 -     x=(a+b)/2; fx=feval(fun,x);
21 -     if abs(fx)<=df*(1+abs(x))*tolx,return;
22 -     elseif fa*fx<0
23 -         b=x;
24 -         fb=fx;
25 -     else
26 -         a=x;
27 -         fa=fx;
28 -     end
29 - end
30 - return
```

il metodo di Newton:

```
Newton.m  X  +
1  function [x,i]= Newton(f,fl,x0,tol,itmax)
2  %   x=newton(f,fl,x0,tol,itmax)
3  %   parametri in ingresso
4  %   f -> funzione in cui si effettua la ricerca degli zeri
5  %   fl -> derivata della funzione f
6  %   x0 -> punto di partenza
7  %   tol -> parametro che indica la tolleranza
8  %   itmax -> numero massimo di iterazioni
9  %   valori di ritorno
10 %   x -> radice della funzione
11 %   i -> numero di iterazioni
12 %   il metodo di newton restituisce la radice di una funzione f
13 %   ed il numero di iterazioni effettuate
14 -   fx = feval(f,x0);
15 -   flx = feval(fl,x0);
16 -   x = x0-fx/flx;
17 -   i = 0;
18 -   while (i<itmax) && (abs(x-x0)>=(1+abs(x))*tol)
19 -       i = i+1;
20 -       x0 = x;
21 -       fx = feval(f,x0);
22 -       flx = feval(fl,x0);
23 -       x = x0-fx/flx;
24 -   end
25 -   if abs(x-x0) >= (1+abs(x))*tol
26 -       error('il metodo non converge')
27 -   end
28 -   return
```

il metodo delle secanti:

```
secanti.m  X  +
1  function [x,i]= secanti(f,fl,x0,tol,itmax)
2  %   x=secanti(f,fl,x0,tol,itmax)
3  %   parametri in ingresso
4  %   f -> funzione in cui si effettua la ricerca degli zeri
5  %   fl -> derivata della funzione f
6  %   x0 -> punto di partenza
7  %   tol -> parametro che indica la tolleranza
8  %   itmax -> numero massimo di iterazioni
9  %   valori di ritorno
10 %   x -> radice della funzione
11 %   i -> numero di iterazioni
12 %   il metodo delle secanti restituisce la radice di una funzione f
13 %   ed il numero di iterazioni effettuate
14 -   fx = feval(f,x0);
15 -   flx = feval(fl,x0);
16 -   x = x0-fx/flx;
17 -   i = 0;
18 -   while (i<itmax) && (abs(x-x0)>=(1+abs(x))*tol)
19 -       i=i+1;
20 -       fx0 = fx;
21 -       fx = feval(f,x);
22 -       x1 = (fx*x0-fx0*x)/(fx-fx0);
23 -       x0 = x;
24 -       x = x1;
25 -   end
26 -   if abs(x-x0) >= (1+abs(x))*tol
27 -       error('il metodo non converge')
28 -   end
29 -   return
```

il metodo delle corde:

```
corde.m  x  +
1  function [x,i]= corde(f,fl,x0,tol,itmax)
2  %   x=corde(f,fl,x0,tol,itmax)
3  %   parametri in ingresso
4  %   f -> funzione in cui si effettua la ricerca degli zeri
5  %   fl -> derivata della funzione f
6  %   x0 -> punto di partenza
7  %   tol -> parametro che indica la tolleranza
8  %   itmax -> numero massimo di iterazioni
9  %   valori di ritorno
10 %   x -> radice della funzione
11 %   i -> numero di iterazioni
12 %   il metodo delle corde restituisce la radice di una funzione f
13 %   ed il numero di iterazioni effettuate
14 -   fx = feval(f,x0);
15 -   flx = feval(fl,x0);
16 -   x = x0-fx/flx;
17 -   i = 0;
18 -   while (i<itmax) && (abs(x-x0)>=(1+abs(x))*tol)
19 -       x0 = x;
20 -       fx = feval (f,x0);
21 -       x = x0-fx/flx ;
22 -       i = i+1;
23 -   end
24 -   if abs(x-x0) >= (1+abs(x))*tol
25 -       error('il metodo non converge')
26 -   end
27 -   return
```

2.2 Esercizio 6

Descrizione: Utilizzare le function del precedente esercizio per determinare una approssimazione della radice della funzione $f(x) = x - e^{-x} \cos(x/100)$, per $tol = 10^{-i}$, $i = 1, 2, \dots, 12$, partendo da $x_0 = -1$. Per il metodo di bisezione, utilizzare $[-1, 1]$, come intervallo di confidenza iniziale. Tabulare i risultati, in modo da confrontare le iterazioni richieste da ciascun metodo. Commentare il relativo costo computazionale.

Soluzione:

tolleranza	bisezione		Newton		secanti		corde	
	radice	iterazioni	radice	iterazioni	radice	iterazioni	radice	iterazioni
10^{-1}	5.000000000000000e-01	2	5.663058026182579e-01	2	5.662928457961099e-01	3	4.021808606806709e-01	2
10^{-2}	5.625000000000000e-01	5	5.671373451065942e-01	3	5.671339926160822e-01	4	5.495185718942192e-01	6
10^{-3}	5.664062500000000e-01	9	5.671373451065942e-01	3	5.671339926160822e-01	4	5.651741531554760e-01	10
10^{-4}	5.671386718750000e-01	13	5.671374702931882e-01	4	5.671374697616252e-01	5	5.670103627779298e-01	15
10^{-5}	5.671386718750000e-01	13	5.671374702931882e-01	4	5.671374697616252e-01	5	5.671232371727831e-01	19
10^{-6}	5.671386718750000e-01	13	5.671374702931882e-01	4	5.671374702931907e-01	6	5.671358764608520e-01	23
10^{-7}	5.671374797821045e-01	23	5.671374702931882e-01	4	5.671374702931907e-01	6	5.671372918143774e-01	27
10^{-8}	5.671374797821045e-01	23	5.671374702931911e-01	5	5.671374702931907e-01	6	5.671374503069613e-01	31
10^{-9}	5.671374704688787e-01	30	5.671374702931911e-01	5	5.671374702931907e-01	6	5.671374689985149e-01	36
10^{-10}	5.671374702360481e-01	33	5.671374702931911e-01	5	5.671374702931911e-01	7	5.671374701482120e-01	40
10^{-11}	5.671374702942558e-01	35	5.671374702931911e-01	5	5.671374702931911e-01	7	5.671374702769563e-01	44
10^{-12}	5.671374702942558e-01	35	5.671374702931911e-01	5	5.671374702931911e-01	7	5.671374702913732e-01	48

Nella tabella sono riportate le radici ricavate con ogni metodo e il relativo numero di iterazioni necessarie per ottenere tale risultato per ogni $tol = 10^{-i}$, $i = 1, 2, \dots, 12$.

Il metodo di bisezione ed il metodo delle corde hanno una valutazione di funzione e convergenza lineare mentre il metodo di Newton ha due valutazioni di funzioni, una per la funzione e una per la sua derivata, e convergenza quadratica. Da quanto detto, consegue che col metodo di Newton si ha una maggiore velocità nella ricerca della radice ma si ha anche un costo computazionale maggiore per singola iterazione. Il metodo delle secanti ha una valutazione di funzione, quindi un costo computazionale minore del metodo di Newton, e una convergenza ≈ 1.618 , quindi ha una velocità maggiore dei metodi di bisezione e delle secanti.

2.3 Esercizio 7

Descrizione: Calcolare la molteplicità della radice nulla della funzione $f(x) = x^2 \cdot \sin(x^2)$. Confrontare, quindi, i metodi di Newton, Newton modificato, e di Aitken, per approssimarla per gli stessi valori di tol del precedente esercizio (ed utilizzando il medesimo criterio di arresto), partendo da $x_0 = 1$. Tabulare e commentare i risultati ottenuti.

Soluzione:

La molteplicità m della radice nulla della funzione $f(x) = x^2 \cdot \sin(x^2)$ é $m = 4$

il metodo di Newton modificato:

```
NewtonMod.m x +
1  function [x, i] = NewtonMod(f, df, m, x0, tol, itmax)
2  %   x=NewtonMod(f,df,m,x0,tol,itmax)
3  %   parametri in ingresso
4  %   f -> funzione in cui si effettua la ricerca degli zeri
5  %   df -> derivata della funzione f
6  %   m -> molteplicità della radice
7  %   x0 -> punto di partenza
8  %   tol -> parametro che indica la tolleranza
9  %   itmax -> numero massimo di iterazioni
10 %   valori di ritorno
11 %   x -> radice della funzione
12 %   i -> numero di iterazioni
13 %   il metodo di newton restituisce la radice di una funzione f
14 %   ed il numero di iterazioni effettuate
15 - i=1;
16 - fx=feval(f,x0);
17 - dfx=feval(df,x0);
18 - x=x0-m*(fx/dfx);
19 - while(i<itmax && abs(x - x0) >= tol*(1+abs(x)))
20 -     i=i+1;
21 -     x0=x;
22 -     fx=feval(f,x0);
23 -     dfx=feval(df,x0);
24 -     x=x0-m*(fx/dfx);
25 - end
26 - if (abs(x - x0) >= tol*(1+abs(x)))
27 -     error('il metodo non converge');
28 - end
29 - return
```

il metodo di aitken:

```
aitken.m  x  +
1  function [x,i] = aitken(f, fl, x0, tol, itmax)
2  %   x=aitken(f,fl,x0,tol,itmax)
3  %   parametri in ingresso
4  %   f -> funzione in cui si effettua la ricerca degli zeri
5  %   fl -> derivata della funzione f
6  %   x0 -> punto di partenza
7  %   tol -> parametro che indica la tolleranza
8  %   itmax -> numero massimo di iterazioni
9  %   valori di ritorno
10 %   x -> radice della funzione
11 %   i -> numero di iterazioni
12 %   il metodo di aitken restituisce la radice di una funzione f
13 %   ed il numero di iterazioni effettuate
14 -   i=0;
15 -   x=x0;
16 -   vai=1;
17 -   while(i<itmax) && vai
18 -       i=i+1;
19 -       x0=x;
20 -       fx=feval(f,x0);
21 -       flx=feval(fl,x0);
22 -       x1=x0-fx/flx;
23 -       fx=feval(f,x1);
24 -       flx=feval(fl,x1);
25 -       x=x1-fx/flx;
26 -       x=(x*x0-x1^2)/(x-2*x1+x0);
27 -       vai = abs(x-x0)>tol*(1+abs(x));
28 -   end
29 -   if vai, error('il metodo non converge'),end
30 -   return
```

tolleranza	Newton		NewtonMod		Aitken	
	radice	iterazioni	radice	iterazioni	radice	iterazioni
10^{-1}	3.84317880607062e-01	2	0	3	6.4929e-19	3
10^{-2}	2.8805139309385e-02	11	0	3	6.4929e-19	3
10^{-3}	2.883766303035e-03	19	0	3	6.4929e-19	3
10^{-4}	2.887022508866152e-04	27	0	3	0	4
10^{-5}	2.890282391459780e-05	35	NaN	4	0	4
10^{-6}	2.893545954951113e-06	43	NaN	4	0	4
10^{-7}	2.896813203496438e-07	51	NaN	4	0	4
10^{-8}	2.900084141256734e-08	59	NaN	4	0	4
10^{-9}	2.903358772397679e-09	67	NaN	4	0	4
10^{-10}	2.906637101089656e-10	75	NaN	4	0	4
10^{-11}	2.909919131507756e-11	83	NaN	4	0	4
10^{-12}	2.913204867831785e-12	91	NaN	4	0	4

CHAPTER 3

SISTEMI LINEARI E NON LINEARI

3.1 Esercizio 8

***Descrizione:** 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:

```

1  function [LU, p] = palu(A)
2  % function [LU, p] = palu(A)
3  % parametri
4  % A -> matrice quadrata non singolare
5  %
6  % funzione che, data una matrice n*n, non singolare, la fattorizza in A=
7  % tramite pivoting parziale. restituisce le informazioni dei fattori
8  % memorizzati in A e il vettore che memorizza le permutazioni effettuate
9  LU=A;
10 [m,n]=size(LU);
11 if m~=n, error('matrice non quadrata');end
12 p=1:n;
13 for i=1:n-1
14     [mi,ki]=max(abs(A(i:n,i)));
15     if mi==0, error('matrice singolare');end
16     ki=ki+i-1;
17     if ki>i
18         LU([i ki],:)=LU([ki i],:);
19         p([i ki])=p([ki i]);
20     end
21     LU(i+1:n,i)=LU(i+1:n,i)/LU(i,i);
22     LU(i+1:n,i+1:n)= LU(i+1:n,i+1:n)-LU(i+1:n,i)*LU(i,i+1:n);
23 end
24 return

```

3.2 Esercizio 9

Descrizione: 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** $x = \text{lusolve}(LU, p, b)$. Curare particolarmente la scrittura e l'efficienza della function.

Svolgimento:

```

lusolve.m  x  +
1  function x = lusolve(LU,p,b)
2      % x=lusolve(LU,p,b)
3      % parametri
4      % LU-> matrice nxn fattorizzata LU
5      % p->vettore delle permutazioni
6      % b->vettore dei termini noti
7      % Data una matrice nxn fattorizzata LU,  p vettore delle permutazioni e
8      % b di lunghezza n, risolve Ax=b, cioè LUx=b Ux=y e Ly=b
9      [m,n]= size (LU);
10     lb = length (b);
11     if abs(m-n) >0 || abs(lb-n) >0
12         error ("dati in ingresso errati");
13     end
14     b=b(p);
15     x=b(:);
16     for j=2:n
17         x(j:n)=x(j:n)-LU(j:n,j-1)*x(j-1);
18     end
19     for j=n:-1:1
20         x(j)=x(j)/LU(j,j);
21         x(1:j -1) =x(1:j -1) -LU(1:j -1,j)*x(j);
22     end
23     return

```

3.3 esercizio 10

Descrizione: Scaricare la function *cremat* che crea sistemi lineari $n \times n$ la cui soluzione é il vettore $v = (1 \cdots n)^T$. Eseguire, quindi, lo script Matlab (vedere esercizio) per testare le function dei precedenti esercizi. Confrontare i risultati ottenuti con quelli attesi, e dare una spiegazione esauriente degli stessi.

Soluzione:

Una volta scaricata la funzione *cremat*

```

cremat.m  x  +
1  function [A,b] = cremat(n,k,simme)
2  %
3  %   [A,b] = cremat(n,k,simme)   Crea una matrice A nxn ed un termine noto b,
4  %                               in modo che la soluzione del sistema lineare
5  %                               A*x=b sia x = [1,2,...,n]^T.
6  %                               k non ve lo dico a cosa serve.
7  %                               simme, se specificato, crea una matrice
8  %                               simmetrica e definita positiva.
9  %
10 - if nargin==1
11 -     sigma = 1/n;
12 - else
13 -     sigma = 10^(-k);
14 - end
15 - rng(0);
16 - [q1,r1] = qr(rand(n));
17 - if nargin==3
18 -     q2 = q1';
19 - else
20 -     [q2,r1] = qr(rand(n));
21 - end
22 - A = q1*diag([sigma 2/n:1/n:1])*q2;
23 - x = [1:n]';
24 - b = A*x;
25 - return

```


ed eseguito il seguente script

```
n = 10;
x = zeros(n,15);
for i = 1:15
    [A,b] = cremat(n,i);
    [LU,p] = palu(A);
    x(:,i) = lusolve(LU,p,b);
end
```

otteniamo i seguenti risultati

```
disp(x(1:10,1:5))
1.0000    1.0000    1.0000    1.0000    1.0000
2.0000    2.0000    2.0000    2.0000    2.0000
3.0000    3.0000    3.0000    3.0000    3.0000
4.0000    4.0000    4.0000    4.0000    4.0000
5.0000    5.0000    5.0000    5.0000    5.0000
6.0000    6.0000    6.0000    6.0000    6.0000
7.0000    7.0000    7.0000    7.0000    7.0000
8.0000    8.0000    8.0000    8.0000    8.0000
9.0000    9.0000    9.0000    9.0000    9.0000
10.0000   10.0000   10.0000   10.0000   10.0000
```

```
disp(x(1:10,6:10))
1.0000    1.0000    1.0000    1.0000    1.0000
2.0000    2.0000    2.0000    2.0000    2.0000
3.0000    3.0000    3.0000    3.0000    3.0000
4.0000    4.0000    4.0000    4.0000    4.0000
5.0000    5.0000    5.0000    5.0000    5.0000
6.0000    6.0000    6.0000    6.0000    6.0000
7.0000    7.0000    7.0000    7.0000    7.0000
8.0000    8.0000    8.0000    8.0000    8.0000
9.0000    9.0000    9.0000    9.0000    9.0000
10.0000   10.0000   10.0000   10.0000   10.0000
```

```
disp(x(1:10,11:15))
1.0000    0.9999    1.0022    1.0060    0.8926
2.0000    2.0002    1.9935    1.9825    2.3131
3.0000    2.9999    3.0021    3.0056    2.8997
4.0000    4.0002    3.9931    3.9815    4.3314
5.0000    4.9995    5.0174    5.0466    4.1649
6.0000    6.0003    5.9914    5.9769    6.4133
7.0000    7.0002    6.9934    6.9823    7.3175
8.0000    8.0000    7.9994    7.9983    8.0300
9.0000    9.0002    8.9940    8.9839    9.2881
10.0000   10.0001    9.9976    9.9936   10.1138
```

notiamo che i risultati ottenuti coincidono con i risultati attesi per k_{12} , quindi

abbiamo dei risultati perturbati.

Questo vuol dire che per il condizionamento del problema per le matrici

$$\frac{\|\Delta x\|}{\|x\|} \leq k(A) \cdot \left(\frac{\|\Delta b\|}{\|b\|} + \frac{\|\Delta A\|}{\|A\|} \right) \quad (3.1)$$

abbiamo un numero di condizionamento di A molto piccolo e quindi una matrice ben condizionata per $1 \leq k < 12$ ed un numero di condizionamento di A $\gg 1$ e quindi una matrice mal condizionata per $k \geq 12$

3.4 Esercizio 11

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

Soluzione:

```

myqr.m  x  +
1  function QR=myqr(A)
2  % QR=myqr(A)
3  % parametri
4  % A -> matrice da fattorizzare QR
5  % funzione che fattorizza QR una matrice A
6  QR=A;
7  [m,n]=size(QR);
8  if m<n,error('il numero di righe è minore del numero di colonne'),end
9  for i=1:n
10     alfa=norm(QR(i:m,i));
11     if alfa==0,error('la matrice A non ha rango massimo'),end
12     if QR(i,i)>=0, alfa=-alfa;end
13     v1 = QR(i,i)-alfa;
14     QR(i,i)=alfa;
15     QR(i+1:m,i)=QR(i+1:m,i)/v1;
16     beta = -v1/alfa;
17     QR(i:m,i+1:n)=QR(i:m,i+1:n)-(beta*[1;QR(i+1:m,i)])*...
18         ([1 QR(i+1:m,i)']*QR(i:m,i+1:n));
19 end
20 return

```

3.5 Esercizio 12

Descrizione: 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 $x = \text{qrsolve}(QR, b)$. Curare particolarmente la scrittura e l'efficienza della function.

Svolgimento:

```

qrsolve.m  x  +
1  function x= qrsolve(A,b)
2  % x=qrsolve(A,b)
3  % parametri
4  % A -> Matrice fattorizzata QR
5  % b -> vettore dei termini noti
6  % la funzione risolve il sistema lineare Ax=b dove A è
7  % una matrice fattorizzata QR
8  [m,n]=size(A);
9  x=b;
10 if length(b)~=m,error('dati inconsistenti');end
11 for i=1:n
12     v=[1; A(i+1:m,i)];
13     beta = 2/(v'*v);
14     x(i:m)=x(i:m)-(beta*(v'*x(i:m)))*v;
15 end
16 x=x(1:n);
17 for i=n:-1:1
18     x(i)=x(i)/A(i,i);
19     if i>1
20         x(1:i-1)=x(1:i-1)-x(i)*A(1:i-1,i);
21     end
22 end
23 return

```

3.6 esercizio 13

Descrizione: Scaricare la function *cremat1* che crea sistemi lineari $m \times n$, con $m \geq n$, la cui soluzione (nel senso dei minimi quadrati) \tilde{x} il vettore $x = (1 \cdots n)^T$. Eseguire, quindi, lo script Matlab (vedere esercizio) per testare le function dei precedenti esercizi.

Soluzione:

una volta scaricata la function *cremat1*

```

cremat.m  X  +
1  function [A,b] = cremat(n,k,simme)
2  %
3  %   [A,b] = cremat(n,k,simme)   Crea una matrice A nxn ed un termine noto b,
4  %                               in modo che la soluzione del sistema lineare
5  %                               A*x=b sia x = [1,2,...,n]^T.
6  %                               k non ve lo dico a cosa serve.
7  %                               simme, se specificato, crea una matrice
8  %                               simmetrica e definita positiva.
9  %
10 - if nargin==1
11 -     sigma = 1/n;
12 - else
13 -     sigma = 10^(-k);
14 - end
15 - rng(0);
16 - [q1,r1] = qr(rand(n));
17 - if nargin==3
18 -     q2 = q1';
19 - else
20 -     [q2,r1] = qr(rand(n));
21 - end
22 - A = q1*diag([sigma 2/n:1/n:1])*q2;
23 - x = [1:n]';
24 - b = A*x;
25 - return

```

```
for n = 5:10
    xx = [1:n]';
    for m = n:n+10
        [A,b] = cremat1(m,n);
        QR = myqr(A);
        x = qrsolve(QR,b);
        disp([m n norm(x-xx)])
    end
end
```

ed aver eseguito lo script `end`

otteniamo i seguenti risultati

n	m	norm(x-xx)
5	5	0.1556e ⁻¹²
6	5	0.0215e ⁻¹²
7	5	0.0130e ⁻¹²
8	5	0.0249e ⁻¹²
9	5	0.0053e ⁻¹²
10	5	0.0117e ⁻¹²
11	5	0.0097e ⁻¹²
12	5	0.0040e ⁻¹²
13	5	0.0056e ⁻¹²
14	5	0.0054e ⁻¹²
15	5	0.0047e ⁻¹²
6	6	0.0539e ⁻¹²
7	6	0.0129e ⁻¹²
8	6	0.0261e ⁻¹²
9	6	0.0170e ⁻¹²
10	6	0.0137e ⁻¹²
11	6	0.0210e ⁻¹²
12	6	0.0128e ⁻¹²
13	6	0.0106e ⁻¹²
14	6	0.0056e ⁻¹²
15	6	0.0084e ⁻¹²
16	6	0.0066e ⁻¹²
7	7	0.0275e ⁻¹²
8	7	0.0536e ⁻¹²
9	7	0.0093e ⁻¹²
10	7	0.0215e ⁻¹²
11	7	0.0265e ⁻¹²
12	7	0.0188e ⁻¹²
13	7	0.0104e ⁻¹²
14	7	0.0236e ⁻¹²

n	m	norm(x-xx)
15	7	0.0165e ⁻¹²
16	7	0.0077e ⁻¹²
17	7	0.0158e ⁻¹²
8	8	0.0941e ⁻¹²
9	8	0.0227e ⁻¹²
10	8	0.0414e ⁻¹²
11	8	0.0324e ⁻¹²
12	8	0.0290e ⁻¹²
13	8	0.0140e ⁻¹²
14	8	0.0121e ⁻¹²
15	8	0.0378e ⁻¹²
16	8	0.0151e ⁻¹²
17	8	0.0115e ⁻¹²
18	8	0.0136e ⁻¹²
9	9	0.0703e ⁻¹²
10	9	0.0712e ⁻¹²
11	9	0.0590e ⁻¹²
12	9	0.0692e ⁻¹²
13	9	0.0222e ⁻¹²
14	9	0.0149e ⁻¹²
15	9	0.0435e ⁻¹²
16	9	0.0285e ⁻¹²
17	9	0.0165e ⁻¹²
18	9	0.0218e ⁻¹²
19	9	0.0263e ⁻¹²
10	10	0.0613e ⁻¹²
11	10	0.1235e ⁻¹²
12	10	0.0683e ⁻¹²
13	10	0.0270e ⁻¹²
14	10	0.0481e ⁻¹²
15	10	0.0395e ⁻¹²
16	10	0.0555e ⁻¹²
17	10	0.0236e ⁻¹²
18	10	0.0193e ⁻¹²
19	10	0.0219e ⁻¹²
20	10	0.0188e ⁻¹²

CHAPTER 4

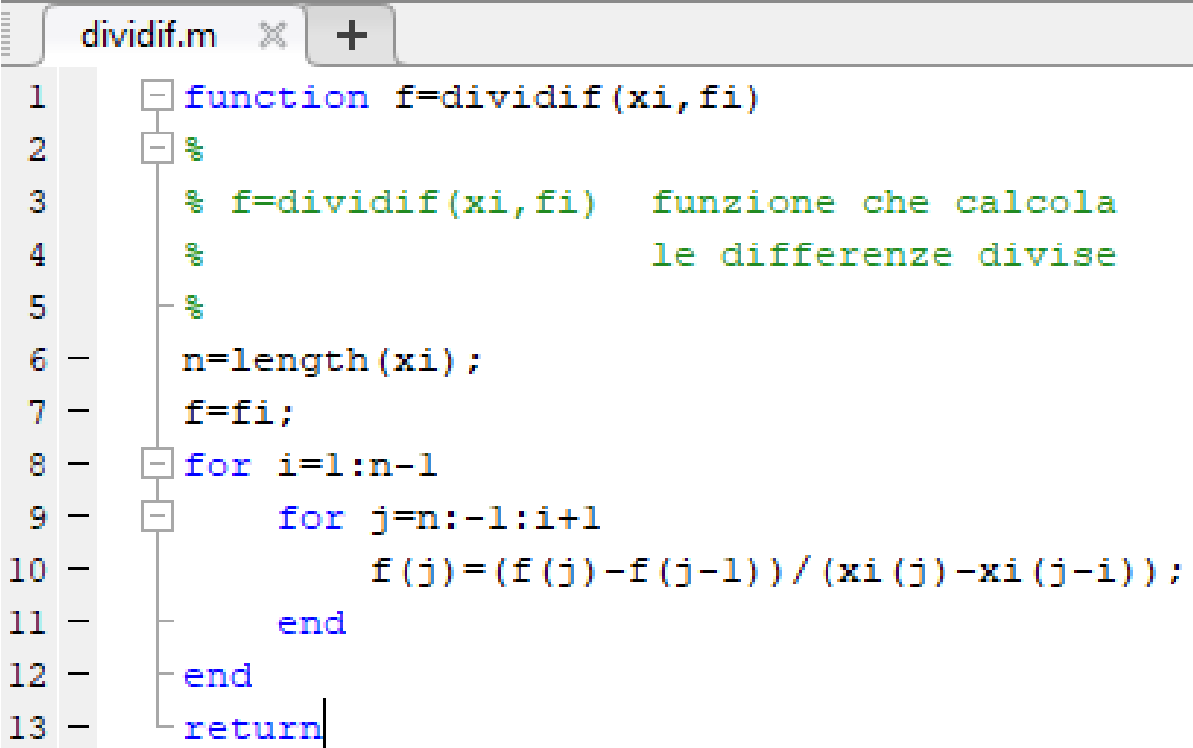
APPROSSIMAZIONE DI FUNZIONI

4.1 Esercizio 14

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

Soluzione:

```
NewtonInterp.m  x  +
1  function y = NewtonInterp(xi,fi,x)
2  %
3  %   y = newton(xi,fi,x) Calcolo il polinomio
4  %                           interpolante le coppie (xi,fi)
5  %                           nelle ascisse x
6  n=length(xi);
7  if length(fi)~=n, error('dati inconsistenti'); end
8  for i=1:n-1
9      for j=i+1:n
10         if xi(i)==xi(j),error('ascisse non distinte');
11         end
12     end
13 end
14 f=dividif(xi,fi);
15 y=f(n);
16 for i=n-1:-1:1
17     y=y.*(x-xi(i))+f(i);
18 end
19 return
```



```
1  function f=dividif(xi,fi)
2  %
3  % f=dividif(xi,fi)  funzione che calcola
4  %                  le differenze divise
5  %
6  n=length(xi);
7  f=fi;
8  for i=1:n-1
9      for j=n:-1:i+1
10         f(j)=(f(j)-f(j-1))/(xi(j)-xi(j-i));
11     end
12 end
13 return
```

4.2 Esercizio 15

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

Soluzione:

```

Hermite.m
1  function y = Hermite(xi,fi,dfi,x)
2  %
3  %   y = Hermite(xi,fi,dfi,x)
4  %
5  %                               Calcolo il polinomio
6  %                               interpolante le coppie (xi,fi)
7  %                               nelle ascisse x
8  n=length(xi);
9  if length(fi)~=n, error('dati inconsistenti'); end
10 for i=1:n-1
11     for j=i+1:n
12         if xi(i)==xi(j), error('ascisse non distinte');
13         end
14     end
15 fh=zeros(2*n);
16 xh=zeros(2*n);
17 for i=1:n
18     fh(2*i-1)=fi(i);
19     fh(2*i)=dfi(i);
20     xh(2*i-1)=xi(i);
21     xh(2*i)=xi(i);
22 end
23 fh=dividifH(fh,xh);
24 y=fh(2*n);
25 for i=2*n-1:-1:1
26     y=y.*(x-xh(i))+fh(i);
27 end
28 return

```

```
dividifH.m  ✕  +
1  -  function f = dividifH(f,x)
2  -  %    f=dividifH(f,x)
3  -  %    calcola le differenze divise secondo il
4  -  %    polinomio interpolante di Hermite
5  -  n=length(x)-1;
6  -  for i=n:-2:3
7  -      f(i)=(f(i)-f(i-2))/(x(i)-x(i-1));
8  -  end
9  -  for j=2:n
10 -      for i=n+1:-1:j+1
11 -          f(i)=(f(i)-f(i-1))/(x(i)-x(i-j));
12 -      end
13 -  end
14 -  return
```

4.3 Esercizio 16

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

Soluzione:

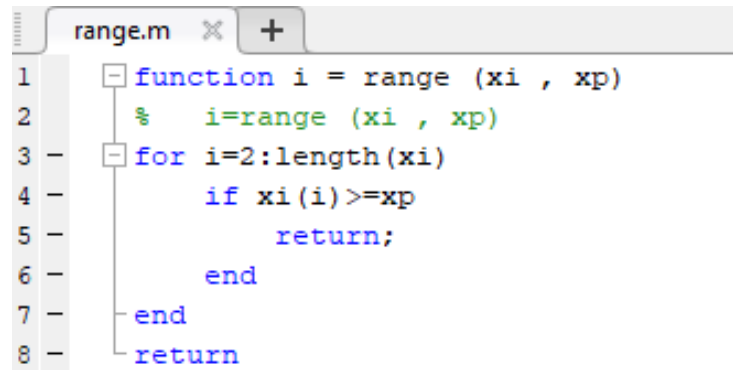
```
naturalSpline.m  X  +
1  function s = naturalSpline(x,y,xp)
2  %   s = naturalSpline(x,y,xp)
3  %   funzione per valutare le ascisse tramite spline naturale
4  %   parametri
5  %   x-> ascisse di interpolazione
6  %   y-> valori della funzione in x
7  %   xp-> punti di interpolazione
8  n=length(x);
9  hi=x(2:n) - x(1:n-1);
10 divdif=(y(2:n)-y(1:n-1))./hi;
11 divdif=6*((divdif(2:end) - divdif(1:end-1))./(x(3:end)-x(1:end-2)));
12 sub=(hi(2:end-1))./(hi(1:end-2)+hi(2:end-1));
13 sup=(hi(2:end-1))./(hi(2:end-1)+hi(3:end));
14 m=tridiag(sub,sup,divdif);
15 m = [0,m,0];
16 ri = zeros (1, n);
17 qi = zeros (1, n);
18 for i = 2:n
19     ri(i-1) = y(i-1) - (( hi(i-1) ^2) /6) * m(i-1);
20     qi(i-1) =(y(i)-y(i-1))/ hi(i-1) - (hi(i-1) /6) * (m(i) - m(i-1));
21 end
22 s=evalSpline(ri,qi,xp,x,m,hi);
23 return
```

```

tridiag.m  x  +
1  function m = tridiag(sub,sup,divdif)
2  %   m = tridiag(sub,sup,divdif)
3  %   funzione che restituisce una matrice tridiagonale
4  -   n=length(divdif);
5  -   l=zeros(1,n);
6  -   v=zeros(n);
7  -   v(1)=2;
8  -   for i = 2:n
9  -       v(i)=2-l(i)*sup(i-1);
10 -       l(i)=sub(i-1)/v(i-1);
11 -   end
12 -   y=zeros(n);
13 -   y(1)=divdif(1);
14 -   for j= 2:n
15 -       y(j)=divdif(j)-l(j)*y(j-1);
16 -   end
17 -   m(n)=y(n)/v(n);
18 -   for k = n-1:-1:1
19 -       m(k)=(y(k)-sup(k)*m(k+1))/v(k);
20 -   end
21 -   return

evalSpline.m  x  +
1  function s=evalSpline(ri,qi,xp,xi,m,hi)
2  % s=evalSpline(ri,qi,xp,xi,m,hi)
3  -   n = length (xp);
4  -   s = zeros (n ,1)';
5  -   for j = 1 : n
6  -       i = range (xi , xp(j));
7  -       s(j) = ( (( xp(j)-xi(i-1) )^3*m(i) + (( xi(i)-xp(j))^3)*m(i-1) )/ ...
8  -       (6* hi(i-1)) ) + qi(i-1) *( xp(j)-xi(i-1) ) + ri(i-1);
9  -   end
10 -   return

```



The image shows a MATLAB script editor window with a tab labeled 'range.m'. The code is as follows:

```
1 function i = range (xi , xp)
2 % i=range (xi , xp)
3 for i=2:length(xi)
4     if xi(i)>=xp
5         return;
6     end
7 end
8 return
```


CHAPTER 5

FORMULE DI QUADRATURA

5.1 esercizio 22

***Descrizione:** Scrivere due functions che implementino efficientemente le formule adattative dei trapezi e di Simpson.*

Soluzione:

```
trapad.m  X  +
1  function I2 = trapad(fun,a,b,tol,fa,fb)
2  % I2 = trapad(fun,a,b,tol,fa,fb)
3  % parametri
4  % fun->funzione
5  % a,b->intervallo
6  % tol->tolleranza
7  % calcola un' approssimazione dell' integrale definito di fun(x)
8  % da a a b con tolleranza tol ( default 1e -5).
9  x1=(a+b)/2;
10 if nargin<=4
11     fa=feval(fun,a);
12     fb=feval(fun,b);
13     if nargin==3
14         tol=1e-5;
15     end
16 end
17 f1=feval(fun,x1);
18 h=(b-a)/2;
19 I1=h*(fa+fb);
20 I2=I1/2+h*f1;
21 err=abs(I2-I1)/3;
22 if err>tol
23     I2=trapad(fun,a,x1,tol/2,fa,f1)+...
24         +trapad(fun,x1,b,tol/2,f1,fb);
25 end
26 return
```

```
simpad.m  x  +
1  function I2 = simpad(fun,a,b,tol,fa,fb,f1)
2  % I2 = simpad(fun,a,b,tol,fa,fb)
3  % parametri
4  % fun->funzione
5  % a,b->intervallo
6  % tol->tolleranza
7  % calcola un' approssimazione dell' integrale definito di fun(x)
8  % da a a b con tolleranza tol ( default 1e-5).
9  x1=(a+b)/2;
10 if nargin<=4
11     fa=feval(fun,a);
12     fb=feval(fun,b);
13     f1=feval(fun,x1);
14     if nargin==3
15         tol=1e-5;
16     end
17 end
18 h=(b-a)/6;
19 I1=h*(fa+4*f1+fb);
20 x2=(a+x1)/2;
21 x3=(x1+b)/2;
22 f2=feval(fun,x2);
23 f3=feval(fun,x3);
24 I2 = 0.5*h*(fa+4*f2+2*f1+4*f3+fb);
25 err=abs(I2-I1)/15;
26 if err>tol
27     I2=simpad(fun,a,x1,tol/2,fa,f1,f2)+...
28         +simpad(fun,x1,b,tol/2,f1,fb,f3);
29 end
30 return
```

5.2 esercizio 23

Descrizione: Sapendo che

$$I(x) = \int_0^{\arctan(30)} (1 + \tan^2(x)) dx = 30 \quad (5.1)$$

tabulare il numero dei punti richiesti dalle formule composite dei trapezi e di Simpson per approssimare $I(f)$ con tolleranze $\text{tol} = 10^{-i}$, con $i = 2, \dots, 8$

Soluzione:

per prima cosa é stato necessario effettuare una modifica al metodo dei trapezi adattivo e al metodo di Simpson adattivo per fare in modo che ritornassero il numero di punti

```
trapadcount.m  x +
1  function [I2,punti] = trapadcount(fun,a,b,tol,fa,fb)
2  % I2 = trapadcount(fun,a,b,tol,fa,fb)
3  % parametri
4  % fun->funzione
5  % a,b->intervallo
6  % tol->tolleranza
7  % calcola un' approssimazione dell' integrale definito di fun(x)
8  % da a a b con tolleranza tol ( default 1e-5).
9  persistent count;
10 x1=(a+b)/2;
11 if nargin<=4
12     fa=feval(fun,a);
13     fb=feval(fun,b);
14     count=0;
15     if nargin==3
16         tol=1e-5;
17     end
18 end
19 count=count+1;
20 f1=feval(fun,x1);
21 h=(b-a)/2;
22 I1=h*(fa+fb);
23 I2=I1/2+h*f1;
24 err=abs(I2-I1)/3;
25 if err>tol
26     I2=trapadcount(fun,a,x1,tol/2,fa,f1)+...
27         +trapadcount(fun,x1,b,tol/2,f1,fb);
28 end
29 punti=count+2;
30 return
```

```

simpadcount.m  x  +
1  function [I2,punti] = simpadcount(fun,a,b,tol,fa,fb,fl)
2  % I2 = simpadcount(fun,a,b,tol,fa,fb)
3  % parametri
4  % fun->funzione
5  % a,b->intervallo
6  % tol->tolleranza
7  % calcola un' approssimazione dell' integrale definito di fun(x)
8  % da a a b con tolleranza tol ( default 1e-5).
9  persistent count;
10 x1=(a+b)/2;
11 if nargin<=4
12     fa=feval(fun,a);
13     fb=feval(fun,b);
14     fl=feval(fun,x1);
15     count=0;
16     if nargin==3
17         tol=1e-5;
18     end
19 end
20 count=count+3;
21 h=(b-a)/6;
22 I1=h*(fa+4*fl+fb);
23 x2=(a+x1)/2;
24 x3=(x1+b)/2;
25 f2=feval(fun,x2);
26 f3=feval(fun,x3);
27 I2 = 0.5*h*(fa+4*f2+2*fl+4*f3+fb);
28 err=abs(I2-I1)/15;
29 if err>tol
30     I2=simpadcount(fun,a,x1,tol/2,fa,fl,f2)+...
31         +simpadcount(fun,x1,b,tol/2,fl,fb,f3);
32 end
33 punti=count+2;
34 return

```

poi si é eseguito il seguente script per visualizzare i risultati

```

f = @(x) 1 + (tan(x)).^2;
a = 0;
b = atan(30);
for i=2:8
    [I2, counter] = simpadcount(f, a, b, 10^-i);
    disp(counter)
    err = abs(30 - I2);
    disp(err)
    [I2, counter] = trapadcount(f, a, b, 10^-i);
    disp(counter)
    err = abs(30 - I2);
    disp(err)
end

```

ottenendo così i risultati della seguente tabella

tolleranza	Simpson		trapezi	
	#punti	errore	#punti	errore
10 ²	71	2.3725e-03	375	4.7645e-03
10 ³	113	6.1653e-04	1181	5.7372e-04
10 ⁴	203	5.0167e-05	3687	5.2639e-05
10 ⁵	371	2.2207e-06	11883	5.4546e-06
10 ⁶	635	3.9818e-07	37273	5.6355e-07
10 ⁷	1145	3.5653e-08	116747	5.2552e-08
10 ⁸	2045	3.5877e-09	375793	5.5077e-09

CHAPTER 6

CALCOLO DEL GOOGLE PAGERANK

6.1 esercizio 24

***Descrizione:** Scrivere una function che implementi efficientemente il metodo delle potenze.*

Soluzione:

```

potenze.m  X  +
1  function [l1,x1] = potenze(A,tol,itmax)
2  % [f1,x1] = potenze(A,tol,itmax)
3  % parametri
4  % A-> matrice
5  % tol->tolleranza
6  % itmax-> numero massimo di iterazioni
7  % funzione che implementa il metodo delle potenze per calcolare
8  % l'autovalore dominante e il corrispettivo autovettore
9  [m,n]=size(A);
10 if m~=n,error('dati inconsistenti'),end
11 if nargin<=2
12     if nargin<=1
13         tol=1e-6;
14     else
15         if tol>= 0.1 || tol<=0, error('tolleranza non valida');end
16     end
17     itmax = ceil (- log10 ( tol))*n;
18 end
19 x = rand(n, 1);
20 l1=0;
21 for k = 1: itmax
22     x1=x/norm(x);
23     x=A*x1;
24     l0=l1;
25     l1=x'*x1;
26     err=abs(l1-l0);
27     if err<=tol*(1+abs(l1)),break,end
28 end
29 if err>tol*(1+abs(l1)),error('convergenza non ottenuta');end
30 return

```

6.2 Esercizio 26

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

Soluzione:

Splitting generico:

```

splitting.m  x  +
1  function x = splitting(b,A,Msolve,tol)
2  % x= splitting(b,A,Msolve,tol)
3  % parametri
4  % b -> vettore colonna dei termini noti
5  % A -> Matrice passata
6  % Msolve -> funzione di Jacobi o di GaussSaider
7  % tol -> tolleranza
8  % la funzione calcola il sistema lineare Ax=b utilizzando un generico
9  % splitting che viene passato tramite il parametro Msolve
10
11  [n,m]=size(A);
12  if(n~=m || n~=length(b)),error ('dati inconsistenti'),end
13  itmax=ceil(-log10(tol))*m;
14  x=zeros(n,1);
15  tolb=tol*norm(b,inf);
16  for i=1:itmax
17      r=A*x-b;
18      nr=norm(r,inf);
19      if nr<=tolb,break,end
20      u=Msolve(r,A);
21      x=x-u;
22  end
23  if nr>tolb, error('tolleranza non raggiunta'),end
24  return

```

6.3 Esercizio 27

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

Soluzione:

il metodo di Jacobi:

```
Jacobi.m  X  +
1  function y = Jacobi(x,A)
2  % y = Jacobi(x,A)
3  % parametri
4  % x -> colonna vettore dei termini noti
5  % A -> matrice nxn
6  % funzione per la risoluzione tramite Jacobi del sistema
7  D=diag(A);
8  y=x./D;
9  return
```

il metodo di GaussSeidel:

```
GaussSeidel.m  X  +
1  function y = GaussSeidel(x,A)
2  % y = GaussSeidel(x,A)
3  % parametri
4  % x -> colonna vettore dei termini noti
5  % A -> matrice nxn
6  % funzione per la risoluzione tramite GaussSeidel del sistema
7  n=length(x);
8  y=x;
9  for i=1:n
10     y(i)=y(i)/A(i,i);
11     y(i+1:n)=y(i+1:n)-y(i)*A(i+1:n,i);
12 end
13 return
```