Codici Matlab/Octave

Stefano De Marchi
Dipartimento di Matematica,
Università di Padova

novembre 2011

Sono qui raccolti i codici citati nel testo *Appunti di Calcolo Numerico - con codici in Matlab/Octave*, I Edizione 2011, pp. 320, ISBN: 9788874884735 Editrice Esculapio-Bologna, ed usati per produrre alcuni dei risultati presentati nel testo. Questi codici sono perfettibili. Sono comunque da considerarsi un' utile traccia per coloro che per la prima volta utilizzano Matlab/Octave. Una nota riguarda la compatibilità tra Matlab e Octave. Una tra le tante piccole differenze tra Matlab e Octave riguarda l'uso degli apici (') (in Matlab) e dei doppi apici (") (in Octave) nei comandi di visualizzazione di strighe. Un'altra differenza riguarda il simbolo di commento, che in Matlab è il % mentre in Octave è #. Per scelta i codici qui presentati girano in Matlab senza alcuna sostituzione dei predetti caratteri. Si rimanda al manuale di Octave, disponibilile in formato .pdf con la distribuzione installata, per la compatibilità con Matlab.

Stefano De Marchi
Dipartimento di Matematica
Università di Padova.

Padova, novembre 2011

Indice dei Capitoli

Capitolo 1

Ricerca di zeri di funzione

1.1 Ricerca di zeri di funzione

```
function [sol,k]=bisezione(a,b,tol)
%-----------------------------------
% Metodo di bisezione
% La funzione fun.m descrive la
% funzione di cui si cerca la radice
%-----------------------------------
% Inputs
%  a,b : estremi dell'intervallo
%  tol : tolleranza
% Output
%  sol : la soluzione cercata
%  k   : numero d'iterazioni fatte
%---------------------------------
if fun(a)*fun(b) > 0
    error('L'' intervallo non contiene alcuna radice');
elseif
    abs(fun(a)*fun(b) ) < tol
    error('Uno degli estremi e'' gia'' sulla radice ')
else
    a0=a; b0=b; k=0;
    disp('Numero iterazioni da fare a priori : ')
    ceil((log(abs(b0-a0))-log(tol))/log(2))

while abs(b-a)>tol*abs(b),
    m=(a+b)/2;
    if abs(fun(m))< tol
      disp('La radice cercata e' : ') m
      break;
```

```
    elseif fun(m)*fun(a)<0,
        b=m;
    else
        a=m;
    end
  k=k+1;
end
end
disp('La radice cercata e' : '); sol=(a+b)/2

disp('Numero iterazioni effettuate : '); k
return



function [x0, niter, flag]=MetIterazioneFunz(x0, tol, kmax)
%--------------------------------------
% Metodo d'iterazione funzionale
%--------------------------------------
% Inputs
% x0: guess iniziale
% tol: tolleranza
% kmax: numero massimo d'iterazioni
%
% Outputs
% x0: soluzione
% niter: iterazioni fatte
% flag: 1 o 0 per indicare se
%       la convergenza c'e' o meno
%--------------------------------------
x1=g(x0); k=1; flag=1;
while abs(x1-x0) > tol*abs(x1) & k <= kmax
  x0=x1;
  x1=g(x0);
  k=k+1;
end
% Se converge, x0 oppure x1 contengono il valore
% dello zero cercato. Altrimenti, si pone flag=0
% che indica la non convergenza
if (k > kmax)
  flag=0;
end
niter=k-1;
return



function[x,iter,stimaerr,m]=NewtonRadiciMult(f,fder,x0,tol,maxit,varargin)
```

```
% ------------------------------------------------
% Function che determina la radice di f e
% la molteplicita' con il metodo di Newton
% ------------------------------------------------
% inputs  f: funzione; fder: derivata di f
%         x0: valore iniziale
% output  x: radice; iter: iterazioni eseguite
%         stimaerr: stima errore; m: molteplicita'
%
%-----------------------------------------------
m0 = 1; x = x0; iter = 1;

stimaerr= -feval(f,x,varargin{:})/feval(fder,x,varargin{:});

x = x+stimaerr; x1 = x; iter = iter+1;

stimaerr = -feval(f,x,varargin{:})/feval(fder,x,varargin{:});

m = m0+1;

while (abs(stimaerr) > tol) & (iter < maxit) & (abs(m-m0)> m/100)
  m0 = m;
  x = x+stimaerr;
  iter = iter+1;
  stimaerr = -feval(f,x,varargin{:})/feval(fder,x,varargin{:});
  m = (x1-x0)/(2*x1-x-x0);
  x0 = x1;
  x1 = x;
end

stimaerr = stimaerr*m;

while (abs(stimaerr) > tol) & (iter < maxit)
  x = x+stimaerr;
  iter = iter+1;
  stimaerr = -m*feval(f,x,varargin{:})/feval(fder,x,varargin{:});
end
 stimaerr = abs(stimaerr);
  if (stimaerr > tol)
    warning('Impossibile raggiungere la tolleranza richiesta')
    warning('entro il numero massimo di iterazioni consentito.')
  end



function [alfa]=Aitken(g,x0,tol,kmax)
%----------------------------------------
% Metodo di accelerazione di Aitken
%----------------------------------------
```

```
% g e‘ la funzione di iterazione g(x)=x-f(x)
%-----------------------------------------
k=0; x1=g(x0); x2=g(x1);
xnew=x0-(x1-x0)^2/(x2-2*x1+x0);
 while abs(x2-xnew)>tol & k <=kmax)
    x0=xnew;
    x1=g(x0);
    x2=g(x1);
    xnew=x0-(x1-x0)^2/(x2-2*x1+x0);
    k=k+1;
 end
% Al termine, alfa sara' l'ultimo valore di xnew
alfa=xnew;
return
```

Capitolo 2

Soluzione di sistemi lineari

2.1 Soluzione di sistemi lineari con metodi iterativi

```
function [xn,i,err,flag]=Jacobi(A,b,x0,nmax,tol)
%----------------------------------------
% Metodo iterativo di Jacobi per sistemi lineari.
%----------------------------------------
% Inputs:
%  A :     Matrice del sistema
%  b :     Termine noto (vettore riga)
%  x0 :    Vettore iniziale (vettore riga)
%  nmax :  Numero massimo di iterazioni
%  tol :   Tolleranza sul test d'arresto (fra iterate)
%
% Outputs
%  xn :    Vettore soluzione
%  i :     Iterazioni effettuate
%  err:    vettore errori relativi in norma 2
%  flag:   se flag=1 converge altrimenti flag=0
%-------------------------------------------------------
flag=1;
D=diag(diag(A)); J=-inv(D)*(A-D); q=inv(D)*b;
xn=J*x0+q;
i=1;
err(i)=norm(xn-x0)/norm(xn);

while (i<=nmax & err(i)>tol)
   x0=xn;
   xn=J*x0+q;
   i=i+1;
   err(i)=norm(xn-x0)/norm(xn);
```

```
end
if i>nmax,
  disp('** Jacobi non converge nel numero di iterazioni fissato');
  flag=0;
end
```

```
function [sol,iter,err, flag]=GaussSeidel(A,b,x0,kmax,tol)
% ---------------------------------------------
% Funzione che implementa il metodo iterativo
% di Gauss-Seidel
%----------------------------------------------
% Inputs
% A, b: matrice e termine noto, rispettivamente
% x0  : guess iniziale
% tol : tolleranza calcoli
%
% Outputs
% sol : vettore soluzione
% iter: numero delle iterazioni
% err:  vettore errori relativi in norma infinito
% flag: booleano convergenza o meno
%----------------------------------------------
n=length(x0); flag=1;
D=diag(diag(A));
L=tril(A)-D; U=triu(A)-D;
DI=inv(D+L); GS=-DI*U;

disp('raggio spettrale matrice iterazione di Gauss-Seidel');
max(abs(eig(GS))) b1=DI*b; x1=GS*x0+b1; k=1;
err(k)=norm(x1-x0,inf)/norm(x1);

while(err(k)>tol & k<=kmax)
    x0=x1;
    x1=GS*x0+b1;
    k=k+1;
    err(k)=norm(x1-x0,inf)/norm(x1);
end
if iter>kmax,
  disp('** Gauss-Seidel non converge nel numero di iterazioni fissato');
  flag=0;
end
sol=x1; iter=k-1;
```

```
function [xv,iter,flag]=GaussRil(a,f,xin,nmax,toll,omega)
%-------------------------------------------------------
% Metodo di Gauss-Seidel rilassato per sistemi lineari.
% Per omega uguale a 1 si ottiene il metodo di Gauss Seidel
% altrimenti e' il metodo SOR
%-------------------------------------------------------------
% Parametri di ingresso:
%  a :     Matrice del sistema
%  f :     Termine noto (vettore riga)
%  xin :   Vettore iniziale (vettore riga)
%  nmax :  Numero massimo di iterazioni
%  toll :  Tolleranza sul test d'arresto (fra iterate)
%  omega : Parametro di rilassamento
%
% Parametri in uscita
%  xv :    Vettore soluzione
%  iter :  Iterazioni effettuate
%  flag :  se flag=1 converge altrimenti flag=0
%-------------------------------------------------------------

flag=1; [n,m]=size(a); d=diag(a); dm1=ones(n,1)./d; dm1=diag(dm1);
b=eye(size(a))-dm1*a; g=dm1*f; bu=triu(b); bl=tril(b);
%-------------
% Iterazioni
%------------
xv=xin; xn=xv;
i=0;
while i<nmax,
  for j=1:n;
    xn(j)=(1-omega)*xv(j)+omega*(bu(j,:)*xv+bl(j,:)*xn+g(j));
  end;
  if abs(xn-xv)<toll,
     iter=i;
     i=nmax+1;
  else
     dif=[dif;norm(xn-xv)];
     xv=xn;
     i=i+1;
  end,
end if i==nmax,
   disp('** Non converge nel numero di iterazioni fissato')
   flag=0;
end



function [omega0]=SOROmegaZero(d,b,n)
%-------------------------------------
```

```
% Ricerca del valore ottimale per l' SOR
%-------------------------------------------
% Inputs
%  d : vettore elementi diagonale principale
%  b : vettore elementi extradiagonali
%  n : lunghezza vettore d
% Output
%  omega0 : il parametro ottimale w0
%-------------------------------------------
% costruisco la matrice
A=diag(d)-diag(b,1)-diag(b,-1);

d1=diag(d); b1=tril(A)-d1; c1=triu(A)-d1;

w=linspace(.1,1.9,100); for i=1:100
 p(i)=max(abs(eig(inv(d1-w(i)*b1)*((1-w(i))*d1+w(i)*c1))));
end;

plot(w,p);
xlabel(' \omega '); ylabel(' p(H_\omega) ');
title('Raggio spettrale della matrice del metodo SOR');
[mp,imp]=min(p);
omega0=w(imp);
```

Capitolo 3

Autovalori di matrici

3.1 Autovalori di matrici

```
function CerchiGerschgorin(A)
%-----------------------------------------------------
% Costruiamo i cerchi di Gerschgorin di una matrice A
%-----------------------------------------------------
tol=1.e-10; Amod=abs(A); n=max(size(A));
raggi=sum(Amod,2)-diag(Amod); xc=real(diag(A)); yc=imag(diag(A));

% angoli per il disegno dei cerchi
th=[0:pi/100:2*pi]; x=[]; y=[];
 figure(1);
 clf;
 axis equal;
 hold on;

for i=1:n,
    x=[x; raggi(i)*cos(th)+xc(i)];
    y=[y; raggi(i)*sin(th)+yc(i)];
    patch(x(i,:),y(i,:),'red');
end
% disegno il bordo e il centro dei cerchi
for i=1:n,
    plot(x(i,:),y(i,:),'k',xc(i),yc(i),'ok');
end

xmax=max(max(x)); ymax=max(max(y)); xmin=min(min(x));
ymin=min(min(y)); hold off; figure(2);
 clf;
 axis equal;
```

```
 hold on;
%-------------------------------------------------------------------
% I cerchi lungo le colonne... sono quelli della matrice trasposta
%-------------------------------------------------------------------
raggi=sum(Amod)-(diag(Amod))'; x=[]; y=[]; for i=1:n,
    x=[x; raggi(i)*cos(th)+xc(i)];
    y=[y; raggi(i)*sin(th)+yc(i)];
    patch(x(i,:),y(i,:),'green');
end
% disegno il bordo e il centro dei cerchi
for i=1:n,
    plot(x(i,:),y(i,:),'k',xc(i),yc(i),'ok');
end
%Determino il bounding box per il plot ottimale
xmax=max(max(max(x)),xmax); ymax=max(max(max(y)),ymax);
xmin=min(min(min(x)),xmin); ymin=min(min(min(y)),ymin); hold off;
axis([xmin xmax ymin ymax]);
figure(1); axis([xmin xmax ymin ymax]);
return


function [lam,x,iter]=MetPotenze(A,tol,kmax,x0)
%-----------------------------------------------------
% Metodo delle potenze per il calcolo dell'autovalore
% di modulo massimo
%-----------------------------------------------------
% Inputs
% A: matrice,
% tol: tolleranza;
% kmax: numero massimo d'iterazioni
% x0: vettore iniziale
%--
% Outputs
% lam : autovalore di modulo massimo
% x: autvettore corrispondente
% iter: iterazioni effettuate
%---------------------------------------------
x0=x0/norm(x0); lam=x0'*(A*x0); err=tol*abs(lam)+1; iter=0;

while (err > tol*abs(lam) & abs(lam)~=0 & iter<=kmax)
    x=A*x0;
    x=x/norm(x);
    lamnew=x'*(A*x);
    err=abs(lamnew-lam);
    lam=lamnew;
    x0=x;
    iter=iter+1;
end
return
```

```
function [rad]=metBernoulli(c,nmax,tol)
%-------------------------------------------------------
% Metodo di deflazione o di Bernoulli per il calcolo
% degli autovalori di una matrice
%-------------------------------------------------------
% inputs:
%     c: vettore dei coefficienti del polinomio
%  nmax: numero max iterazioni per il metodo delle potenze
%   tol: tolleranza richiesta
% output:
%   rad: vettore delle radici richieste
%-------------------------------------------------------

c1=c(1:length(c)-1)/c(length(c)); %normalizzo
n=length(c1);

% Costruisco la matrice di Frobenius

F(n,:)=-c1; F=F+diag(ones(1,n-1),1);
%-----------------------------------------------
% Applico la funzione MetPotenze per il calcolo
% dell'autovalore di modulo max della matrice F
% a partire da un vettore t0 formato da tutti 1
%-----------------------------------------------
t0=ones(n,1);
[lam1,t1,iter]=MetPotenze(F,tol,namx,t0);
rad(1)=lam1;
%---------------------------------------------------------------
% Calcolo del secondo autovalore-radice per deflazione
% Costruisco la matrice di Householder P t.c. P*t1=(1,0,...,0)
%---------------------------------------------------------------
t12=norm(t1,2);
beta=1/(t12*(t12+abs(t1(1)) ));
v(1)=sign(t1(1))*(abs(t1(1))+t12);
v(2:n)=t1(2:n);
P=eye(n)-beta*v'*v;

F1=P*F*P';

%-------------------------------------------------------------
% Calcolo i rimanenti autovalori per deflazione
% y, la prima volta, indichera' l'autovettore corrispondente
% all'autovalore lam(1).
%-------------------------------------------------------------
for s = 2:n
  m=length(y);
  t12=norm(y,2);
  beta=1/(t12*(t12+abs(y(1))));
```

```
  v(1)=sign(y(1))*(abs(y(1))+t12);
  v(2:m)=y(2:m);
  P=eye(m)-beta*v'*v;

  F1=P*F*P';
  F=F1(2:end,2:end);
  x=rand(m,1);
  [rad(s), y, iter] = MetPotenze(F,tol,nmax,x);
  clear v
end
disp('Controllo usando la funzione roots')
norm(roots(c(end:-1:1))-lam')



function [T,iter]=MetQR(A,tol,kmax)
%-----------------------------------------------------------
% Metodo QR per il calcolo di tutti gli autovalori di una
% matrice data.
%-----------------------------------------------------------

[Q,R]=qr(A); T=Q*R;
%..................................................
% QQ e' la matrice contenente il prodotto comulativo
% delle matrici ortogonali Q_k, ovvero
% QQ=\prod_{k=1}^M Q_k.
%..................................................
QQ=Q; k=1; while convergenzaQR(T,tol)==0 & k <= kmax,
    T1=R*Q;
    [Q,R]=qr(T1);
    QQ=QQ*Q;
    T=Q*R;
    k=k+1;
end
disp('Numero iterazioni ' ) k
% Verifica

disp('Calcolo del residuo diag(T)-eig(A) ' )
norm(diag(T),2)-norm(eig(A),2) iter=k-1;



function nn=convergenzaQR(T,tol)
%-----------------------------------------------------------
% Controlla che gli elementi extradiagonali della matrice
% T sono sotto la tolleranza
%-----------------------------------------------------------
[n1,n2]=size(T); if n1 ~= n2,
```

```
    error('Attenzione .... matrice non quadrata');
    return
end if sum(abs(diag(T,-1))) <= tol,
    nn=1;
else
    nn=0;
end return
```

```
function [T,iter]=MetQRShift(A,tol,kmax)
%---------------------------------------
% Metodo QR con shift
%---------------------------------------
n=size(A,1);
T=hess(A); % forma di Hessenberg di A
iter=1;

for k=n:-1:2,
    I=eye(k);
    while convergenzaQRShift(T,tol,k)==0 & iter <= kmax,
     mu=T(k,k);
     [Q,R]=qr(T(1:k,1:k)-mu*I);
     T(1:k,1:k)=R*Q+mu*I;
     iter=iter+1;
    end
T(k,k-1)=0;
end
```

dove il test di convergenza, si farà implementando una funzione `convergenzaQRShift` per la disuguaglianza

$$|t^{(k)}_{n,n-1}| < \epsilon \left(|t^{(k)}_{n-1,n-1}| + |t^{(k)}_{n,n}| \right) \quad \epsilon \approx \text{eps}. \tag{3.1}$$

```
function [D,iter,phi]=symJacobi(A,tol)
%-------------------------------------------------------------
% Ricerca di tutti gli autovalori di una matrice
% simmetrica con il metodo di Jacobi
%-------------------------------------------------------------------
% Data la matrice simmetrica A e una tolleranza tol, determina,
% mediante il metodo di Jacobi tutti i suoi autovalori che memorizza
% nella matrice diagonale D. Determina inoltre il numero di iterazioni,
% iter, e la quantita' phi che contiene la norma degli elementi
% extra-diagonali e come converge il metodo.
%
% Richiede tre funzioni
```

```
% calcoloCeS         : calcola coseno e seno
% calcoloProdottoGtDG: determina il prodotto G'DG
% phiNorm            : \sqrt(\sum_{i=1:n-1,j=i+1:n}^n a_{ij}^2+a_{ji}^2)
%-------------------------------------------------------------------------
n=max(size(A)); D=A; phiD=norm(A,'fro'); epsi=tol*phiD;
phiD=phiNorm(D); iter=0; [phi]=phiD;

while (phiD > epsi)
    iter=iter+1;
    for p=1:n-1,
        for q=p+1:n
            [c,s]=calcoloCeS(D,p,q);
            D=calcoloProdottoGtDG(D,c,s,p,q);
        end;
    end
    phiD=phiNorm(D);
    phi=[phi; phiD];
end
return
```

Capitolo 4

Interpolazione e approssimazione

4.1 Interpolazione polinomiale

```
function l=lagrai(z,x,i)
%-------------------------------------------------
% Calcola l'i-esimo pol. elementare di Lagrange
%
% Inputs
% z: nodi di interpolazione
% x: punto su cui valutare
% i: indice del polinomio
%
% l: valore del polinomio in x
%-------------------------------------------------
z1=setdiff(z,[z(i)]);
l=prod(x-z1)/prod(z(i)-z1);
return

\bigskip

function l = lagrai_target(z,x,i)
%-------------------------------------------------
% Calcola l'i-esimo pol. elementare di Lagrange
% su un vettore di punti target
%-------------------------------------------------
% z = nodi di interpolazione
% x = vettore (colonna!) di punti target
%     su cui valutare l_i
% i = indice del polinomio
%
% l = vettore dei valori di l_i sui targets
```

```
%-----------------------------------------------
n = length(z); m = length(x);

l = prod(repmat(x,1,n-1)-repmat(z([1:i-1,i+1:n]),m,1),2)/...
prod(z(i)-z([1:i-1,i+1:n])); return
```

```
function leb=CostLebesgue(x)
%-------------------------------------------------------------
% Dato il vettore x di N nodi (ordinato), la funzione calcola
% il valore della costante di Lebesgue per i=2,...,N
%
% Input: x (vettore ordinato dei nodi)
% Output: leb (vettore dei valori della costante di Lebesgue)
%-------------------------------------------------------------
a=x(1); b=x(end); M=1000;
xx=linspace(a,b,M); %sono i punti "target"
N=length(x);

for i=2:N,
 for s=1:i
     l(s,:)=lagrai_target(x,xx,s);
 end
leb(i-1)=norm(l,1);
end
return
```

```
function [b]=DiffDivise(x,y)
%-------------------------------------------------
% Algoritmo delle differenze divise
%-------------------------------------------------
% Inputs
% x: vettore dei punti di interpolazione,
% y: vettore dei valori della funzione.
% Output
% b: vettore riga delle differenze divise
%    b=[b_1,....b_n]
%-------------------------------------------------
n=length(x); b=y;
for i=2:n,
    for j=2:i,
       b(i)=(b(i)-b(j-1))/(x(i)-x(j-1));
    end;
end;
```

```
function p=Horner(d,x,xt)
%-------------------------------------------------
% Questa funzione implementa lo schema di Horner
% per la valutazione del polinomio d'interpolazione
% in forma di Newton su un insieme di punti target
%-------------------------------------------------
% Inputs
% d: vettore delle differenze divise
% x: vettore dei punti d'interpolazione
% xt: vettore dei punti target
%
% Output
% p: il polinomio valutato in tutti i punti
%    target
%-------------------------------------------------
n=length(d);

for i=1:length(xt) p(i)=d(n);
 for k=n-1:-1:1
  p(i)=p(i)*(xt(i)-x(k))+d(k);
 end
end
```

4.1.1 Bsplines

```
function y = bspline(i,k,xi,x)
%-----------------------------------------------------
% Questa funzione valuta la i-esima B-spline
% di ordine k, che usa come nodi xi, nel punto x
%-----------------------------------------------------
% valori iniziali per la relazione di ricorrenza
if (k == 1)
  y = zeros(size(x));
  index = find(xi(i) <= x & x < xi(i+1));
  y(index) = 1;
  return
end
val = (xi(i+k) == xi(i+1));

a =(xi(i+k)-x)/(xi(i+k)-xi(i+1)+val)*(1-val);

val = (xi(i+k-1) == xi(i));

b = (x-xi(i))/(xi(i+k-1)-xi(i)+val)*(1-val);
```

```
y = a.*bspline(i+1,k-1,xi,x)+b.*bspline(i,k-1,xi,x);

% continuita' sull'estremo destro dell'intervallo
index2 = find(x == max(xi) & i == length(xi)-k);
y(index2) = 1;
```

Nota. Le ultime due righe, servono a rendere continue a destra le B-spline e in particolare quando si vuole costruire una base con nodi multipli con molteplicità pari all'ordine. Si veda per questo il paragrafo 5.7.2 in cui i nodi d'interpolazione sono scelti così da soddisfare alle condizioni di Carrasso e Laurent (cfr. [9]).

4.1.2 FFT

```
function [ff]=myFFT(f,p)
%-------------------------------------------
% Dato un vettore f di lunghezza n=p*q
% determina il vettore ff che rappresenta
% la FFT di f (algoritmo tratto dal riferimento
% [1], ovvero K. E. Atkinson pag. 181 e ss)
%-------------------------------------------

disp('Trasformata di Fourier tradizionale'); n=length(f);

for j2=1:n,
    s=0;
    for j1=1:n,
        s=s+f(j1)*exp(-2*pi*i/(n*j2*(j1-1)));
    end
    f1(j2)=s/n;
end

disp('Trasformata di Fourier veloce'); p=2; q=n/p;

for k=1:n,
    s1=0;
    for l=1:p,
        s=0;
        for g=1:q,
          w=exp(2*pi*i*k*(g-1)/q);
          s=s+w*f(l+p*(g-1));
       end
       w1=exp(2*pi*i*k*(l-1)/n);
       s1=s1+(s/q)*w1;
   end
   ff(k)=s1/p;
```

```
end
```

Capitolo 5

Integrazione

5.1 Integrazione

```
function [integral,nv]=simp_ada(aa,bb,epss,f)
%-------------------------------------------------------------
%  Calcolo di un integrale con il metodo di Simpson adattativo
%-------------------------------------------------------------
% Inputs
% aa,bb: estremi dell'intervallo d'integrazione
% epss: tolleranza richiesta
% f: la funzione da integrare
%
% Outputs
% integral: valore approssimato dell'integrale
% nv: numero di valutazioni di funzioni
%-------------------------------------------------------------
NMAXL=100;        % numero massimo livelli
integral=0;       % valore approssimato dell'integrale
max_err=0;        % errore commesso

ff=input('Fattore dilatazione tolleranza = '); i=1;
l(i)=1;           % contatore numero livelli
tol(i)=ff*epss;
%---------------------------Plot della funzione integranda
x=aa:.01:bb; for j=1:length(x),
    y(j)=f(x(j));
end; plot(x,y);
title('Metodo di Simpson adattativo');
hold on;
%-------------------------------------------------------
a(i)=aa; b(i)=bb;
```

```
h(i)=(b(i)-a(i))/2;
min_h=h(i);
m(i)=a(i)+h(i);
fa(i)=f(a(i));
fb(i)=f(b(i));
fm(i)=f(m(i));
s(i)=h(i)/3*(fa(i)+4*fm(i)+fb(i));
nv=3;     % Valutazioni di funzione
while( i > 0),
% ---------------  PLOT dei punti di integrazione
      p(1)=a(i); p(2)=a(i)+h(i)/2; p(3)=a(i)+h(i);
      p(4)=a(i)+1.5*h(i); p(5)=a(i)+2*h(i);
      plot(p,zeros(5,1),'r+');
      fp(1)=f(a(i));fp(2)=f(a(i)+h(i)/2);fp(3)=f(a(i)+h(i));
      fp(4)=f(a(i)+1.5*h(i));fp(5)=f(a(i)+2*h(i));
      plot(p,fp,'g+');
      nv=nv+2;
% ---------------------------------------------
      fd=f(a(i)+h(i)/2);
      fe=f(a(i)+3/2*h(i));
      ss1=h(i)/6*(fa(i)+4*fd+fm(i));
      ss2=h(i)/6*(fm(i)+4*fe+fb(i));
%---------------------------
%  Salvo i dati del livello
%---------------------------
      t1=a(i); t2=fa(i); t3=fm(i);  t4=fb(i);
      t5=h(i); t6=tol(i); t7=s(i); t8=l(i);
      i=i-1;
      if (abs(ss1+ss2-t7) < t6)
         max_err=abs(ss1+ss2-t7);
         integral=integral+ss1+ss2;
         if( t5 < min_h),
             min_h=t5;
         end;
      elseif (t8 >= NMAXL)
          disp('Superato il livello max. STOP! ');
          break;
      else
% Meta' Intervallo dx .
         i=i+1;
         a(i)=t1+t5;
         fa(i)=t3; fm(i)=fe; fb(i)=t4;
         h(i)=t5/2;
         tol(i)=t6/2;
         s(i)=ss2;
         l(i)=t8+1;
%% Meta' Intervallo sx.
         i=i+1;
```

```
         a(i)=t1;
         fa(i)=t2; fm(i)=fd; fb(i)=t3;
         h(i)=h(i-1);
         tol(i)=tol(i-1);
         s(i)=ss1;
         l(i)=l(i-1);
      end;
end;
disp('Valore approssimato dell''integrale'); integral
legend('r+','Punti integr.','g+','Valore funzione'); hold off;
disp('Errore max. = ');
max_err disp('Minimo step usato '); min_h
return
```

```
function [I,errest,x] = trapadatt(func,a,b,tol,varargin)
%-------------------------------------------------------------
%  Calcolo di un integrale con
% il metodo dei trapezi adattativo
%-------------------------------------------------------------
% Inputs
% func: la funzione da integrare
% a,b: estremi dell'intervallo d'integrazione
% tol: tolleranza richiesta
%
% Outputs
% I: valore approssimato dell'integrale
% errest: stima d'errore
% x: vettore contenente i nodi usati
%-------------------------------------------------
if (nargin == 4)
  n = 2;
else
  n = varargin{1};
end
h = (b-a)/(n-1);
x = linspace(a,b,n)';

x(2:end-1) = x(2:end-1)+2*(rand(size(x(2:end-1)))-0.5)*h/10;

weight = h*[0.5,ones(1,length(x)-2),0.5];
I = weight*feval(func,x);
n = 2*n-1; h = h/2; x = linspace(a,b,n)';

x(2:end-1)=x(2:end-1)+2*(rand(size(x(2:end-1)))-0.5)*h/10;

weight=h*[0.5,ones(1,length(x)-2),0.5];
```

```
I2 = weight*feval(func,x); errest = abs(I2-I)/2;

if (errest < tol)
  I = I2;
else
  [Il,errestl,xl] = trapadatt(func,a,a+(b-a)/2,tol/2,n);
  [Ir,errestr,xr] = trapadatt(func,a+(b-a)/2,b,tol/2,n);
  I = Il+Ir;
  errest = errestl+errestr;
  x = union(xl,xr);
end

return
```

5.1.1 Tecnica di Romberg

```
function int_value=romberg(f,a,b,n)
%-------------------------------------------------
% Questa M-function implementa il metodo di Romberg
% per la quadratura numerica partendo dalla
% formula dei trapezi composta (mediante la
% formula (6.61)).
%-------------------------------------------------
% Inputs
% f: funzione d'integrazione
% n: numero livelli
% a,b: estremi intervallo d'integrazione
%
% Output
% int_value: valore dell'integrale
%-----------------------------------------
T=zeros(n,n);
%---------------
% Prima colonna
%---------------
for i=1:n,
  m=2^(i-1); h=(b-a)/m;
  x=linspace(a,b,m+1);
  ff=f(x);
  T(i,1)=(h/2)*(ff(1)+ff(end)+2*sum(ff(2:end-1)));
end;
%--------------------
% Colonne successive
%--------------------
```

```
for k=2:n,
  for i=k:n,
    T(i,k)=(4^(k-1)*T(i,k-1)-T(i-1,k-1))/(4^(k-1)-1);
  end;
end;

disp('Tabella di Romberg'); T
int_value=T(n,n);
```

Capitolo 6

Derivazione numerica

6.1 Derivazione numerica

```
function mydiff(x0)
%---------------------------------------
% Calcola la derivata dell'esponenziale
% in x0 con il rapporto incrementale
% e con differenze finite centrali
% per 50 differenti valori del passo.
% Calcola l'errore relativo dei due
% metodi e li plotta.
%---------------------------------------

% valore esatto della derivata
f1xesatta=exp(x0);

for index=1:50
%---------------------------------------
% METODO 1: differenze finite in avanti
%---------------------------------------
% passo
    h=2^(-index);
% punto "x"
    x=x0+h;
% rapporto incrementale
    f1x(index)=(exp(x)-exp(x0))/h;
% errore relativo 1
    relerr1(index)=abs(f1xesatta-f1x(index))/abs(f1xesatta);
    fprintf('\n \t [x]: %5.5f [h]: %2.2e',x,h);
    fprintf('[rel.err.]: %2.2e',relerr1(index));
```

```
%-------------------------------------
% METODO 2: differenze finite centrali
%-------------------------------------
% punti di valutazione
    xplus=x+h; xminus=x-h;
% approssimazione con differenze finite centrali
    f1x(index)=(exp(xplus)-exp(xminus))/(2*h);
% errore relativo 2
    relerr2(index)=abs(f1xesatta-f1x(index))/abs(f1xesatta);
    fprintf('\n \t [x]: %5.5f [h]: %2.2e',x,h);
    fprintf('[rel.err.]: %2.2e',relerr2(index));
end

% plot degli errori
semilogy(1:50,relerr1,'r-+', 1:50,relerr2,'k-o');
return
```

6.2 Equazione di Van der Pol

Questo M-function, che potremo chiamare `provaVanderP.m`, consente di confrontare alcune delle funzioni Matlab per sistemi stiff `ode15s, ode23s, ode23tb`, sull'equazione di Van der Pol.

```
function provaVanderP(choice)
%-------------------------------------------------------
% provo la soluzione del sistema di Van der Pol
% con varie tecniche
%-------------------------------------------------------
switch choice

case 1:
t0=clock;
[t,u]=ode15s(@vanderp,[0 3000],[2 0],[],1000);
%ODE15S Solve stiff differential equations and DAEs
disp('Tempo richiesto per risolvere il sistema con ode15s')
etime(clock,t0)
disp('Min passo usato= ')
min(diff(t))

plot(t,u(:,1));

case 2:
%----------------------------------------
t0=clock;
[t,u]=ode23s(@vanderp,[0 3000],[2 0],[],1000);
%ODE23S  Solve stiff differential equations, low order method.
disp('Tempo richiesto per risolvere il sistema con ode23s')
etime(clock,t0)
disp('Min passo usato= ')

min(diff(t))
plot(t,u(:,1));

case 3:
%----------------------------------------
t0=clock;
[t,u]=ode23tb(@vanderp,[0 3000],[2 0],[],1000);
%ODE23TB Solve stiff differential equations, low order method.
disp('Tempo richiesto per risolvere il sistema con ode23tb')
etime(clock,t0)
disp('Min passo usato= ')
min(diff(t))
plot(t,u(:,1));
```

```
end
```

Indice analitico

Bibliografia

[1] K. E. Atkinson, *An Introduction to Numerical Analysis*, Second Edition, Wiley, New York, 1989.

[2] R. Bevilacqua, D. Bini, M. Capovani e O. Menchi *Metodi Numerici*, Zanichelli, 1992.

[3] J.-P. Berrut, Lloyd N. Trefethen, *Barycentric Lagrange Interpolation*, SIAM Rev. 46(3) (2004), pp. 501-517.

[4] V. Comincioli, *Analisi numerica: metodi, modelli, applicazioni. E-book*, Apogeo, 2005.

[5] V. Comincioli, *Analisi numerica. Complementi e problemi*, McGraw-Hill Companies, 1991.

[6] M. Conrad e N. Papenberg, *Iterative Adaptive Simpsons and Lobatto Quadrature in Matlab*, TR-2008-012 Mathematics and Computer Science, Emory University, 2008.

[7] P. J. Davis, *Interpolation & Approximation*, Dover Publications Inc., New York, 1975.

[8] C. de Boor, *A Practical Guide to Splines*, Springer-Verlag, New York, 1978.

[9] S. De Marchi, *Funzioni splines univariate*, Forum Ed. Udinese, Seconda ed., 2001 (con floppy).

[10] G. Farin, *Curves and Surfaces for CAGD: A Practical Guide*, Third Edition, Academic Press, San Diego, 1993.

[11] Gautschi, W., Inglese, G., Lower bounds for the condition numbers of Vandermonde matrices, *Numer. Math.*, 52(3) (1988), pp. 241-250.

[12] G. Golub, Charles F. Van Loan *Matrix computation*, The Johns Hopkins University Press, Terza Edizione, 1996.

[13] D. Greenspan, V. Casulli *Numerical Analysis for Applied Mathematics, Science and Engineering*, Addison-Wesley, 1988.

[14] Higham, N. J., The Scaling and Squaring Method for the Matrix Exponential Revisited, *SIAM J. Matrix Anal. Appl.*, 26(4) (2005), pp. 1179-1193.

[15] E. Isaacson, H. Bishop Keller, *Analysis of Numerical Methods*, John Wiley & Sons, New York, 1966.

[16] G. G. Lorentz, *Bernstein Polynomials*, Chelsea Publishing Company, New York, 1986.

[17] Moler, C. B. and C. F. Van Loan, Nineteen Dubious Ways to Compute the Exponential of a Matrix, *SIAM Review 20*, 1978, pp. 801-836.

[18] G. Monegato *Elementi di Calcolo Numerico*, Levrotto&Bella, Torino, 1995.

[19] A. Quarteroni, F. Saleri *Introduzione al Calcolo Scientifico*, Esercizi e problemi risolti in Matlab, Terza Ed., Springer-Verlag, Milano, 2006.

[20] A. Quarteroni, R. Sacco e F. Saleri *Matematica Numerica*, Seconda Ed., Springer-Verlag, Milano, 2004.

[21] T. J. Rivlin, *An Introduction to the Approximation of Functions*, Dover Publications Inc., New York, 1969.

[22] J. Stoer, Bulirsch *Introduction to Numerical Analysis* Ed. Springer-Verlag, Berlin, 1980.