Equazioni nonlineari in Matlab, per Ingegneria dell'Energia
Esercizi e loro risoluzione [1]

A. Sommariva[2]

Abstract

Metodo di Bisezione, metodo di Newton, esercizi.

Ultima revisione: 21 dicembre 2018

1. Metodo di bisezione: esercizi

Esercizio.

1. Si modifichi il programma `bisezione` (cf.[5]) mediante la nuova routine `bisezione2` cosicchè termini le sue iterazioni qualora l'ampiezza dell'ultimo intervallo analizzato sia inferiore a una tolleranza `tollintv` o il residuo pesato sia inferiore a `toll`.
   Osservazione: si aggiunga la variabile `tollintv` agli input della funzione.
2. Si modifichi `demo_bisezione` in `demo_bisezione2` che utilizzi `bisezione2`. In particolare si assegni a `tollintv` il valore di 10^{-6}.
3. Si effettuino entrambi gli esperimenti, aventi quali valori del parametro `esempio` i numeri 1 e 2.

1.1. *Implementazione di* `bisezione2`

```
function [aa,bb,wr,flag]=bisezione2(f,a,b,toll,tollintv,...
    maxit)

% Algoritmo di bisezione, con criterio di arresto sul ...
    residuo pesato e
% ampiezza dell'intervallo.
%
% Dati di ingresso:
% f: funzione (inline function)
% a: estremo sinistro
% b: estremo destro
% toll: tolleranza richiesta per il test del residuo ...
    pesato
% tollintv: tolleranza richiesta per il test sull'amp. ...
    dell'intervallo
% maxit: massimo indice dell'iterata permesso
%
% Dati di uscita:
% aa: sequenza degli estremi di sinistra degli ...
    intervalli [a_k,b_k],
%     immagazzinata in vettore colonna;
% bb: sequenza degli estremi di destra degli intervalli ...
    [a_k,b_k],
%     immagazzinata in vettore colonna;
% wr: sequenza dei residui pesati,immagazzinata in ...
    vettore colonna;
% flag: 0 processo terminato correttamente,
%     1 processo non terminato correttamente.

if b < a
    s=b; b=a; a=s;
end % Aggiusta errori utente.

flag=0;
fa=feval(f,a); fb=feval(f,b);
aa=[a]; bb=[b]; wr=[];

% test: ipotesi bisezione non soddisfatte
if fa*fb > 0, flag=1; return; end

% zero all'estremo iniziale "a".
if fa == 0, aa=[a]; bb=[a]; return; end % a sol.

% zero all'estremo iniziale "b".
if fb == 0, aa=[b]; bb=[b]; return; end % b sol.

if (b-a) < tollintv, c=(a+b)/2; aa=[c]; bb=[c]; end

% iterazioni bisezione
for k=1:maxit
    c=(a+b)/2;  fc=feval(f,c); % punto medio di [a_k,b_k...
     ]
    w=(b-a)/(fb-fa); % peso "w".
    wres=abs(fc*w);  % residuo pesato.
    ampiezza=(b-a)/2; % ampiezza intervallo (a,c)=(c,b).
    wr=[wr; wres];   % aggiorna sequenza residui pesati

    if (wres<toll) |(fc==0) | (ampiezza<tollintv)
        aa=[aa;c]; bb=[bb;c]; return;% OK exit.
    end

    % determinazione intervallo [a_{k+1},b_{k+1}]
    if sign(fc) == sign(fa) % "c" sostituisce "a"
        % aggiorna "aa", "bb", "a", "fa"
        aa=[aa; c]; bb=[bb; b]; a=c; fa=fc;
    else % "c" sostituisce "b"
        % aggiorna "aa", "bb", "b", "fb"
        aa=[aa; a]; bb=[bb; c]; b=c; fb=fc;
    end
end

% se si e' raggiunto questo punto, si sono fatte troppe ...
    iterazioni
flag=1;
```

1.1.1. *Commento a* `bisezione2`

Relativamente a `bisezione2`:

- abbiamo aggiunto `tollintv` come quarta variabile di input.
- all'interno del ciclo-for (cf. [1]) abbiamo modificato

```
if (wres<toll) |(fc==0)
```

in

```
if(wres<toll)|(fc==0)|(ampiezza<tollintv)
```

1.2. *Implementazione di* `demo_bisezione2`

Per quanto riguarda `demo_bisezione2`:

```
function demo_bisezione2

% default
toll=10^(-6);
tollintv=10^(-6);
maxit=1000;
esempio=1;

% esempi.
switch esempio
    case 1 % funzione piatta
        f=inline('exp(x)-2+x');
        a=0; b=1;
        sol=0.4428544010023885;
    case 2
        f=inline('sin(x)-x');
        a=-2; b=3;
        sol=0;
end

% bisezione
[aa,bb,wr,ko]=bisezione2(f,a,b,toll,tollintv,maxit);

% statistiche
fprintf('\n \t soluzione : %1.15e',aa(end));
fprintf('\n \t tolleranza: %1.15e',toll);
fprintf('\n \t numero it.: %4d',length(aa));

if ko == 0
    fprintf('\n \t La procedura e'' terminata ...
     correttamente');
else
    fprintf('\n \t La procedura non e'' terminata ...
     correttamente');
end
fprintf('\n \n');

% ---------------------- plot ----------------------

indici=(1:length(wr))';

% grafico risultati
clf;
semilogy(indici,wr);
hold on;
title('Residuo pesato bisezione'); % titolo
xlabel('Indice'); % etichetta asse x
ylabel('Residuo pesato'); % etichetta asse y
% nome del file da salvare che vari con l'esempio,
% ottenuto concatenando 3 stringhe
nomefile_pdf=strcat('bisezione2_esempio',num2str(esempio...
    ),'.pdf');
% salva figura come pdf.
print(nomefile_pdf,'-dpdf');
hold off;

% ---------- salvataggio risultati su file ----------

% nome file variabile con l'esempio.
nomefile_txt=strcat('bisezione2_esempio',num2str(esempio...
    ),'.txt');
% creazione del file con facolta' di scrittura.
fid=fopen(nomefile_txt,'w');
% dati immagazzinati nella matrice A (si immagazzinino ...
    come vettori riga,
% ma bisogna ricordare che "aa", "bb" sono colonna.
A=[1:length(aa); aa'; bb'];
% scrittura dei dati su file.
fprintf(fid,'\n %3.0f  %1.15e %1.15e',A);
% chiusura file
fclose(fid);
```

1.2.1. *Commento a* `demo_bisezione2`

L'unica differenza con `demo_bisezione` è (oltre alla chiamata della funzione `function demo_bisezione2`) che invece di

```
[aa,bb,wr,ko]=bisezione(f,a,b,toll,maxit);
```

abbiamo

```
[aa,bb,wr,ko]=bisezione2(f,a,b,toll,tollintv,maxit);
```

2. Il metodo di Newton

Esercizio.

1. Aiutandosi con quanto fatto in `bisezione` e con il pseudocodice fornito, si implementi il metodo di Newton (cf.[6]) in Matlab, salvandolo nel file `newtonfun.m`. In particolare, si utilizzi l'intestazione

```
function [xv, fxv, step, flag] = newtonfun (f, f1,...
      x0, toll, maxit)
% Metodo di Newton
%
% Dati di ingresso:
%   f:      funzione
%   f1:     derivata prima
%   x0:     valore iniziale
%   toll:   tolleranza richiesta per il modulo
%     della differenza di due iterate successive
%   maxit:   massimo numero di iterazioni permesse
%
% Dati di uscita:
%   xv:     vettore contenente le iterate
%   fxv:    vettore contenente la valutazione di f
%           in xv
%   step: vettore contenente gli step
%   flag: 0 la derivata prima non si e' annullata.
%         1 la derivata prima si e' annullata,
%         2 eseguite piu' iter. di maxit.
```

2. Si implementi una versione di `newtonfun.m`, diciamo `newtonfun_for.m` che utilizzi un ciclo-for invece di un ciclo while (cf. [2]). A tal proposito si esca per return (cf. [3]) se la derivata prima si annulla in una iterazione ponendo `flag` uguale a 1 o se il valore assoluto dello step è minore della tolleranza ponendo `flag` uguale a 0. Se dopo `maxit` iterazioni il valore assoluto dello step è ancora maggiore o uguale alla tolleranza si ponga `flag` uguale a 2.

3. Utilizzando quale base `demo_newton`, si implementi la routine `demo_newton_for` che testi il metodo di Newton relativamente al calcolo degli zeri di
   (a) $f(x) = \exp(x) - 2 + x$ partendo dal valore iniziale `x0` uguale a 1,
   (b) $f(x) = \sin(x) - x$ partendo dal valore iniziale `x0` uguale a 1.

   utilizzando `newtonfun_for.m`. Si ponga `toll` uguale a 10^{-6}, `maxit` uguale a 1000.

   La demo deve contenere il codice relativo al plot del valore assoluto dello step e della stampa su file
   (a) degli indici della componente del vettore `xv` in formato decimale con 4 cifre intere,
   (b) il vettore `xv` in formato esponenziale con 1 cifra prima della virgola e 15 dopo,
   (c) il valore assoluto dello step `step`, in formato esponenziale con 1 cifra prima della virgola e 15 dopo.
4. Si vedano i risultati ottenuti dal metodo di Newton per ogni singolo esempio. Il numero di iterazioni è inferiore a quello di bisezione?

2.1. *Implementazione di* `newtonfun`

Per quanto riguarda `newtonfun` il codice richiesto è :

```
function [xv, step, flag] = newtonfun (f, f1, x0, toll, ...
     maxit)

% Metodo di Newton, con criterio di arresto dello step.
%
% Dati di ingresso:
%   f:      funzione
%   f1:     derivata prima
%   x0:     valore iniziale
%   toll:   tolleranza richiesta per il modulo
%     della differenza di due iterate successive
%   maxit:   massimo numero di iterazioni permesse
%
% Dati di uscita:
%   xv:     vettore riga contenente le iterate
%   step: vettore riga contenente i moduli degli step
%   flag: 0 la derivata prima non si e' annullata.
%         1 la derivata prima si e' annullata,
%         2 eseguite piu' iter. di maxit.

% inizializzazione output
flag=0; step=toll+1; xv=x0;

n=1;
while (step(end) ≥ toll) & (n < maxit) & (flag == 0)
    if f1(xv(n)) == 0
        flag=1;
    else
        % calcolo nuovo step
        s=-f(xv(n))/f1(xv(n));
        % calcolo nuova iterazione
        xv(n+1)=xv(n)+s;
        % calcolo valore assoluto step.
        step=[step abs(s)];
        % aggiornamento indice di iterazione
        n=n+1;
    end
end

if (step(end) ≥ toll) & (flag == 0)
    % siccome si esce dal while, e' falso l'asserto
    % (step(end) ≥ toll) & (n < maxit) & (flag == 0)
    % ma (step(end) ≥ toll) & (flag == 0) e' vero.
    % Necessariamente si esce per troppe iterazioni.
    flag=2;
end
```

2.1.1. *Commento a* `newtonfun`

Il codice ricalca molto quello scritto nel pseudocodice. L'unica sostanziale differenza consiste nella gestione del vettore degli `step`, che nel pseudocodice era uno scalare mentre in `newtonfun` si richiede sia un vettore.

Abbiamo modificato

```
while (step ≥ toll) & (n < maxit) & (flag == 0)
```

in

```
while (step(end) ≥ toll) & (n < maxit) & (flag == 0)
```

Il valore `step(end)` è l'ultimo valore del vettore `step`.

Di seguito in

```
      % calcolo nuovo step
      s=-f(x(n))/f1(x(n));
      % calcolo nuova iterazione
      x(n+1)=x(n)+s;
      % calcolo valore assoluto step.
      step=[step; abs(s)];
      n=n+1;
```

calcoliamo il nuovo step $-f(x_n)/f^1(x_n)$ e lo immagazziniamo in `s`.

Calcoliamo il nuovo valore proposto dal metodo di Newton e lo salviamo in `x(n+1)`.

Quindi aggiorniamo lo step, aggiungendo al vettore `step` una nuova componente che ha valore `abs(s)`.

Risulta interessante la porzione di codice

```
if (step(end) ≥ toll) & (flag == 0)
    % siccome si esce dal while, e' falso l'asserto
    % (step(end) ≥ toll) & (n < maxit) & (flag == 0)
    % ma (step(end) ≥ toll) & (flag == 0) e' vero.
    % Necessariamente si esce per troppe iterazioni.
    flag=2;
end
```

Usciti dal ciclo while abbiamo varie possibilità .

- Risulta che è vero

  `(step(end) < toll) & (flag == 0)`

  e in questo caso il test dell'istruzione condizionale è falso e quindi si esce con la soluzione corretta e `flag` uguale a 0.

- Risulta che è vero

  `(step(end) >= toll) & (flag == 0)`

  e quindi abbiamo fatto troppe iterazioni, per cui si pone `flag` uguale a 2;

- Risulta che `flag` è uguale a 1 e quindi il test dell'istruzione condizionale è falso e quindi non si esce con la soluzione corretta e `flag` uguale a 1.

2.2. *Implementazione di* `newtonfun_for`

Salviamo in `newtonfun_for.m` il seguente codice.

```
function [xv, step, flag] = newtonfun_for (f, f1, x0, ...
    toll, maxit)

% Metodo di Newton, con criterio di arresto dello step.
%
% Dati di ingresso:
%   f:      funzione
%   f1:     derivata prima
%   x0:     valore iniziale
%   toll:   tolleranza richiesta per il modulo
%     della differenza di due iterate successive
%   maxit:   massimo numero di iterazioni permesse
%
% Dati di uscita:
%   xv:     vettore contenente le iterate
%   step: vettore contenente i moduli degli step
%   flag: 0 la derivata prima non si e' annullata.
%         1 la derivata prima si e' annullata,
%         2 eseguite piu' iter. di maxit.

% inizializzazione output
flag=0; step=toll+1; xv=x0;

for n=1:maxit
   if f1(xv(n)) == 0
       flag=1; return;
   else
       % calcolo nuovo step
      s=-f(xv(n))/f1(xv(n));
      % calcolo nuova iterazione
      xv(n+1)=xv(n)+s;
      % calcolo valore assoluto step.
      step=[step; abs(s)];
      if step(end) < toll
          return;
      end
    end
end

% se abbiamo raggiunto questo punto, allora abbiamo ...
    fatto troppe
% iterazioni, visto che non siamo usciti per "return".
flag=2;
```

2.2.1. *Commento a* `newtonfun_for`

- La parte cruciale è la sostituzione del ciclo-while con il ciclo-for. Per uscire in caso di
  - insuccesso con `flag` pari a 1, durante le iterazioni, ovvero per derivata nulla all'ultima iterazione,
  - successo per `step` inferiore alla tolleranza richiesta,

  si usa il `return`. Dopo il `return`, non vengono lette ulteriori righe di codice.
- Se il codice arriva all'ultima riga, significa solo che si sono fatte troppe iterazioni e quindi al `flag` viene assegnato il valore 2.

2.3. *Implementazione* `demo_newton`

Salviamo il file `demo_newton.m` avente quale contenuto

```
function demo_newton

% default
toll=10^(-6);
nmax=1000;
esempio=2;

% esempi.
switch esempio
    case 1 % funzione piatta
        f=inline('exp(x)-2+x');
        f1=inline('exp(x)+1');
        x0=1;
        sol=0.4428544010023885;
    case 2
        f=inline('sin(x)-x');
        f1=inline('cos(x)-1');
        x0=1;
        sol=0;
end

% newton
[xv,step,flag] = newtonfun (f,f1,x0,toll,nmax);

% statistiche
fprintf('\n \t soluzione : %1.15e',xv(end));
fprintf('\n \t tolleranza: %1.15e',toll);

% il numero di iterazioni e' pari alla lunghezza di
% xv meno 1.
fprintf('\n \t numero it.: %4d',length(xv)-1);

if flag == 0
    fprintf('\n \t La procedura e'' terminata ...
     correttamente');
else
    fprintf('\n \t La procedura non e'' terminata ...
     correttamente');
end
fprintf('\n \n');

% ---------------------- plot ----------------------

indici=(1:length(step))';

% grafico risultati
clf;
semilogy(indici,step);
hold on;
title('Step Newton'); % titolo
xlabel('Indice'); % etichetta asse x
ylabel('Modulo dello step'); % etichetta asse y
% nome del file da salvare che vari con l'esempio,
% ottenuto concatenando 3 stringhe
nomefile_pdf=strcat('newton_esempio',num2str(esempio),'....
    pdf');
% salva figura come pdf.
print(nomefile_pdf,'-dpdf');
hold off;

% ---------- salvataggio risultati su file ----------

% nome file variabile con l'esempio.
nomefile_txt=strcat('newton_esempio',num2str(esempio),'....
    txt');
% creazione del file con facolta' di scrittura.
fid=fopen(nomefile_txt,'w');
% dati immagazzinati nella matrice A
A=[(1:length(xv)); xv; step];
% scrittura dei dati su file.
fprintf(fid,'\n %3.0f  %1.15e  %1.2e',A);
% chiusura file
fclose(fid);
```

Abbiamo usato quanto scritto in `demo_bisezione`, ma anche

- modificato il nome della function;
- aggiunto la funzione `f1` relativa alle derivate prime;
- tolto i valori di `a`, `b` e immesso i valori iniziali richiesti in `x0`;
- inserito la chiamata alla function `newtonfun`;
- modificato le statistiche, utilizzando `xv(end)` e `n`;
- nell'istruzione condizionale abbiamo messo `flag` al posto di `ko`;
- nei grafici, abbiamo sostituito la variabile `step` al posto di `xv`.

Per quanto riguarda la descrizione del grafico e il salvataggio dei dati, il codice è essenzialmente uguale a `demo_bisezione`, solo che

- non ha a che fare con i vettori `aa`, `bb` ma con `xv`,
- non ha a che fare con il vettore del residuo pesato `wr`, ma con quello dei valori assoluti dello step `step`.

Nota. 2.1. *Come visto precedentemente, nella parte di salvataggio su file, i vettori riga vengono salvati su una matrice* `A` *che viene utilizzata per descrivere i dati su file. La stranezza è che in generale* `A` *è una matrice con* 3 *righe e diciamo* M *colonne, ma nella tabella viene stampata una lista di dati composta da* M *righe e* 3 *colonne.*

La routine `demo_newton_for` è essenzialmente uguale, solo che invece di

```
[xv,step,n,flag] = newtonfun (f,f1,x0,toll,maxit);
```

si scrive

```
[xv,step,n,flag] = newtonfun_for (f,f1,x0,toll,maxit);
```

Quali risultati

1. se poniamo `esempio` uguale a 1, ottenendo

```
>> demo_newton

   soluzione : 4.428544010023886e-01
   tolleranza: 1.000000000000000e-06
   numero it.:    5
   La procedura e' terminata correttamente

>>
```

e e il grafico in figura che suggerisce una convergenza quadratica;
Il file `newton_esempio1.txt` generato dal codice risulta

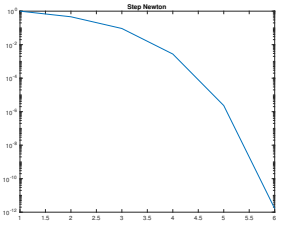

Figura 1: Grafico degli step per il primo esempio, suggerendo una convergenza quadratica

```
   1 1.000000000000000e+00 1.00e+00
   2 5.378828427399902e-01 4.62e-01
   3 4.456167485265452e-01 9.23e-02
   4 4.428567246451099e-01 2.76e-03
   5 4.428544010040325e-01 2.32e-06
   6 4.428544010023886e-01 1.64e-12
```

2. se poniamo `esempio` uguale a 1, ottenendo

```
>> demo_newton

   soluzione : 1.499003612735834e-06
   tolleranza: 1.000000000000000e-06
   numero it.:   33
   La procedura e' terminata correttamente

>>
```

e il grafico in figura che suggerisce una convergenza lineare.

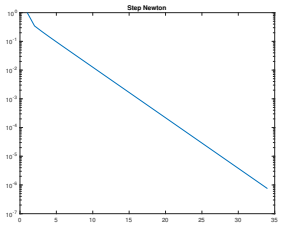

Figura 2: Grafico degli step per il secondo esempio, suggerendo una convergenza lineare

Il file `newton_esempio2.txt` generato dal codice risulta

```
   1  1.000000000000000e+00  1.00e+00
   2  6.551450720424304e-01  3.45e-01
   3  4.335903683634927e-01  2.22e-01
   4  2.881484008925012e-01  1.45e-01
   5  1.918323121506386e-01  9.63e-02
   6  1.278096675607083e-01  6.40e-02
   7  8.518323360286406e-02  4.26e-02
   8  5.678195278661637e-02  2.84e-02
   9  3.785260078111326e-02  1.89e-02
  10  2.523446453501505e-02  1.26e-02
  11  1.682279781087122e-02  8.41e-03
  12  1.121514564046264e-02  5.61e-03
  13  7.476748086523549e-03  3.74e-03
  14  4.984494080301871e-03  2.49e-03
  15  3.322994677536352e-03  1.66e-03
```

```
  16   2.215329377336683e-03   1.11e-03
  17   1.476886130694148e-03   7.38e-04
  18   9.845907179818797e-04   4.92e-04
  19   6.563938011773572e-04   3.28e-04
  20   4.375958645594350e-04   2.19e-04
  21   2.917305751797622e-04   1.46e-04
  22   1.944870494640509e-04   9.72e-05
  23   1.296580324165062e-04   6.48e-05
  24   8.643868936790834e-05   4.32e-05
  25   5.762579478000866e-05   2.88e-05
  26   3.841719669159640e-05   1.92e-05
  27   2.561146824771111e-05   1.28e-05
  28   1.707431201620750e-05   8.54e-06
  29   1.138287797203844e-05   5.69e-06
  30   7.588596587592925e-06   3.79e-06
  31   5.059065812115213e-06   2.53e-06
  32   3.372690698315936e-06   1.69e-06
  33   2.248438823081083e-06   1.12e-06
  34   1.499003612735834e-06   7.49e-07
```

Nota. 2.2. *Si sottolinea che utilizzando* `demo_newton_for`, *si ottengono risultati identici.*

3. Il metodo di punto fisso

Si supponga di voler calcolare un certo x^* tale che $x = \phi(x)$. Il metodo di *punto fisso* definisce, partendo da un certo $x^{(0)}$ la successione, detta delle *approssimazioni successive*, $x^{(k+1)} = \phi(x^{(k)})$.

Basandosi sulla routine `newtonfun.m`, definire una routine `punto_fisso.m` che

- risolva il problema di punto fisso mediante la successione delle approssimazioni successive,
- salvi tutte le iterate eseguite nel vettore `xv` la cui k componente corrisponde alla $k-1$-sima iterata;
- si ponga quale prima componente di `xv` il valore di ingresso `x0`;
- si salvino in `step` i valori assoluti dello step al variare delle iterazioni;
- arresti il processo quando il valore assoluto dello step è minore di una tolleranza `toll`,
- se vengono eseguite più di `maxit` iterazioni si esca comunque dalla procedura ponendo `flag=1` altrimenti esca con `flag=0`.

Si utilizzi quale intestazione

```
function [xv, step, flag] = puntofisso (phi, x0, toll, ...
    maxit)

% Metodo di punto fisso, con criterio di arresto dello ...
    step.
%
% Dati di ingresso:
%   phi:    funzione di punto fisso (risolve x=phi(x)
%   x0:     valore iniziale
%   toll:   tolleranza richiesta per il modulo
%     della differenza di due iterate successive
%   maxit:   massimo numero di iterazioni permesse
%
% Dati di uscita:
%   xv:     vettore riga contenente le iterate
%   step:   vettore riga contenente i moduli degli step
%   flag:   0 termina correttamente
%           1 eseguite piu' iter. di maxit.
```

3.1. *Risoluzione*

Salviamo nel file `punto_fisso.m` il seguente codice.

```
function [xv, step, flag] = puntofisso (phi, x0, toll, ...
    maxit)

% Metodo di punto fisso, con criterio di arresto dello ...
    step.
%
% Dati di ingresso:
%   phi:    funzione di punto fisso (risolve x=phi(x)
%   x0:     valore iniziale
%   toll:   tolleranza richiesta per il modulo
%     della differenza di due iterate successive
%   maxit:   massimo numero di iterazioni permesse
%
% Dati di uscita:
%   xv:     vettore riga contenente le iterate
%   step:   vettore riga contenente i moduli degli step
%   flag:   0 termina correttamente
%           1 eseguite piu' iter. di maxit.

% inizializzazione output
flag=0; step=toll+1; xv=x0;

n=1;
while (step(end) ≥ toll) & (n < maxit) & (flag == 0)
        % calcolo nuova iterazione
        xv(n+1)=phi(xv(n));
        % calcolo valore assoluto step.
        step=[step abs(xv(end)-xv(end-1))];
        % aggiornamento indice di iterazione
        n=n+1;
end

if (step(end) ≥ toll) & (flag == 0)
    % siccome si esce dal while, e' falso l'asserto
    % (step(end) ≥ toll) & (n < maxit) & (flag == 0)
    % ma (step(end) ≥ toll) & (flag == 0) e' vero.
    % Necessariamente si esce per troppe iterazioni.
    flag=1;
end
```

La routine è sostanzialmente la precedente.

- Tra gli input, invece di `f` e `f1`, si assegna `phi`.
- Non serve controllare che si annullino derivate.
- Il valore assoluto dello step è

  `abs(x(end)-x(end-1))`.

4. Equazione di Colebrook

Si definisca una function `demo_colebrook` che risolvi l'equazione di Colebrook

$$\frac{1}{\sqrt{\lambda}} = -2\log_{10}\left(\frac{e}{3.51\cdot d} + \frac{2.52}{N_R\sqrt{\lambda}}\right)$$

determinando la soluzione positiva di

$$x = -2\log_{10}\left(\frac{e}{3.51\cdot d} + \frac{2.52\cdot x}{N_R}\right)$$

Si supponga sia

- $e = 1$,
- $d = 1$,
- $N_R = 1000$,

e si risolva il problema con il metodo delle approssimazioni successive, utilizzando quali parametri

- `x0` $= 1$,
- `toll` $= 10^{-8}$,
- `maxit` $= 100000$,

Quali statistiche si riportino i valori

- `flag` con solo una cifra prima della virgola, nessuna dopo la virgola, in formato decimale;
- la soluzione finale `xv(end)` con solo una cifra prima della virgola, 15 dopo la virgola, in formato esponenziale;
- il numero di iterazioni `iter` con solo 6 cifre prima della virgola, nessuna dopo la virgola, in formato decimale;
- l'ultimo step `step(end)` con solo una cifra prima della virgola, 15 dopo la virgola, in formato esponenziale;
- il valore `abs(xv(end)-phi(xv(end)))` con solo una cifra prima della virgola, 15 dopo la virgola, in formato esponenziale.

4.1. Risoluzione

Salviamo in `demo_colebrook` il seguente codice.

```
function demo_colebrook

% in questa demo studiamo l'equazione di colebrook
% 1/sqrt(lambda)=-2*log10( (e/3.51*d) + 2.52/(NR*sqrt(...
    lambda)) )
% dove e,d,NR sono fissati dal problema e "lambda" e l'...
    incognita

% ---- parametri punto fisso ----
x0=1;
toll=10^(-8);
maxit=100000;

% ---- parametri colebrook ----
e=1;
d=1;
NR=10^1;

% ---- metodo scelto ----

% risolviamo x=-2*log10( (e/3.51*d) + 2.52*x/NR ) (per x...
    ≥ 0) e poi
% poniamo x=1/sqrt(lambda) ovvero lambda=1/x^2.
phi=@(x) -2*log10( (e/3.51*d) + 2.52*x/NR );
[xv, step, flag] = puntofisso (phi, x0, toll, maxit);


% ---- statistiche ----

fprintf('\n \t valore flag: %1.0f',flag);
fprintf('\n \t soluzione  : %1.15e',xv(end));
fprintf('\n \t iterazioni : %8.0f',length(xv));
fprintf('\n \t step       : %1.1e',step(end));
fprintf('\n \t abs(x-phi(x)): %1.1e',abs(xv(end)-phi(xv...
    (end))));
fprintf('\n \n');
```

L'unica cosa da notare è che il logaritmo in base 10 viene calcolato mediante `log10`.

Lanciato il codice da command-window ricaviamo

```
>> demo_colebrook

   valore flag: 0
   soluzione  : 6.810642512103869e-01
   iterazioni :       26
   step       : 9.7e-09
   abs(x-phi(x)): 4.7e-09

>> % si osservi che da lambda=1/x^2 la soluzione e' ...
     1/(6.810642512103869e-01)
>> format long e; lambda=1/(6.810642512103869e-01)
lambda =
     1.468290250476076e+00
>>
```

Bibliografia

[1] Mathworks, Ciclo For, https://www.mathworks.com/help/matlab/ref/for.html
[2] Mathworks, Ciclo While, https://www.mathworks.com/help/matlab/ref/while.html
[3] Mathworks, Return, https://www.mathworks.com/help/matlab/ref/return.html
[4] Wikipedia, Ciclo For, https://it.wikipedia.org/wiki/Ciclo_for
[5] Wikipedia, Metodo della Bisezione, https://it.wikipedia.org/wiki/Metodo_della_bisezione
[6] Wikipedia, Metodo delle Tangenti, https://it.wikipedia.org/wiki/Metodo_delle_tangenti