Analisi Numerica
Relazioni di Laboratorio

Indrjo Dedej

Ultima revisione: 25 giugno 2023.

Indice

Nota 0.1. È stato usato *GNU Octave* versione 8.2.0 invece di *Matlab*.

Nota 0.2. Se A è il nome di una matrice, allora scriviamo $A_{i,j}$ per indicare l'entrata di A alla i-esima riga e j-esima colonna.

1 *Metodi diretti per sistemi lineari*

Esercizio 1.1. Scrivere una funzione `tri_solve` che prenda in input una matrice triangolare A e un vettore b e calcoli la soluzione del sistema lineare $Ax = b$. Tale funzione deve gestire sia il caso triangolare superiore che triangolare inferiore (si usino eventualmente le funzioni `istriu` e `istril`).

Una matrice quadrata A di ordine n si dice *triangolare superiore* (rispettivamente, *inferiore*) qualora $A_{i,j} = 0$ per ogni $i, j \in \{1, \dots, n\}$ con $i > j$ (rispettivamente, $i < j$). Un sistema triangolare inferiore è quindi fatto quindi di equazioni

$$\sum_{j=1}^{i} A_{i,j} x_j = b_i \quad \text{per } i \in \{1, \dots, n\}.$$

Si verifica immediatamente che la soluzione $x \coloneqq \begin{bmatrix} x_1 & \cdots & x_n \end{bmatrix}^T$ di questo sistema è tale che

$$x_1 = \frac{b_1}{A_{1,1}}$$
$$x_i = \frac{1}{A_{i,i}} \left(b_i - \sum_{j=1}^{i-1} A_{i,j} x_j \right)$$

La funzione qui sotto prende una matrice triangolare inferiore A e un vettore b e restituisce la soluzione del sistema $Ax = b$.

```
function [x] = solve_lower_triangular(A,b)
  n = length(b);
  x(1) = b(1)/A(1,1);
```

```
  for i = 2:n
    x(i) = (b(i) - A(i,1:i-1) * x(1:i-1)')/A(i,i);
  end
  x = x';
end
```

Invece, la soluzione $x := \begin{bmatrix} x_1 & \cdots & x_n \end{bmatrix}^T$ di un sistema triangolare superiore

$$\sum_{j=n-i+1}^{n} A_{i,j} x_j = b_i \quad \text{per } i \in \{1, \dots, n\}$$

è data da

$$x_n = \frac{b_n}{A_{n,n}}$$

$$x_{n-i+1} = \frac{1}{A_{n-i+1,n-i+1}} \left(b_n - \sum_{j=n-i+2}^{i-1} A_{i,j} x_j \right)$$

La funzione che risolve $Ax = b$ nel caso in cui A sia triangolare superiore è:

```
function [x] = solve_upper_triangular(A,b)
  n = length(b);
  x(n) = b(n)/A(n,n);
  for i = 1:n
    i = n-i+1;
    x(i) = (b(i) - A(i,i+1:n) * x(i+1:n)')/A(i,i);
  end
  x = x';
end
```

Pertanto la funzione che risolve un sistema triangolare $Ax = b$ è la seguente.

```
function [x] = solve_triangular(A,b)
  if istril(A)
    [x] = solve_lower_triangular(A,b);
  elseif istriu(A)
    [x] = solve_upper_triangular(A,b);
  else
    disp("the matrix isn't triangular!");
    return;
  end
end
```

Esercizio 1.2. Scrivere due funzioni, `lu_nopiv` e `lu_piv`, che calcolino la fattorizzazione LU con e senza pivoting di una matrice, rispettivamente.

L'algoritmo di eliminazione gaussiana consente, presa una matrice invertibile A di ordine n, di trasformare il sistema

$$Ax = b$$

in uno equivalente

$$Ux = b'$$

dove U è triangolare superiore. Il passaggio da A ad U avviene attraverso una matrice L triangolare inferiore con gli elementi sulla diagonale principale uguali ad 1:

$$A = LU.$$

Le entrate della matrice L sono ottenute durante l'algoritmo di Gauss. Qui sotto ci sono le funzioni che ad una matrice A restituisce la fattorizzazione LU, la prima senza e la seconda con pivotazione.

```
function [L U] = lu_nopiv(A)
  % l'ordine della matrice quadrata A
  n = rows(A);
  % All'inizio, la matrice triangolare inferiore è
  % l'identità, ma di volta in volta verrà aggiornata.
  L = eye(n);
  % Facciamo una copia di A, la quale verrà modificata
  % passo dopo passo. In realtà, non ce ne sarebbe il
  % bisogno, in quanto basta alterare A direttamente.
  U = A;
  for i = 1:n
    for j = i+1:n
      % Questi sono i coefficienti ottenuti durante
      % il MEG. La cosa importante è che U(i, i) =/= 0.
      % La pivotazione evita ciò, ma in questa funzione
      % non la consideriamo.
      L(j,i) = U(j,i)/U(i,i);
      % Le righe successive all'i-esima sono ottenute
      % sottraendo un opportuno multiplo dell'i-esima.
      U(j,:) = U(j,:)-L(j,i)*U(i,:);
    end
  end
end
```

La fattorizzazione LU con pivotazione coinvolge un'altra matrice:

$$PA = LU$$

dove P è ottenuta dall'identità permutandone le righe.

```
function [L U P] = lu_piv(A)
  n = rows(A);
  L = eye(n);
  U = A;
  P = eye(n);
  for i = 1:n
    % Ricerca del primo j per cui |A(j,i)| è massimo
    m = 0; k = i;
    for j = i:n
      if abs(A(j,i)) > m
        k = j; m = abs(A(j,i));
      end
    end
    L([i k],1:i-1) = L([k i],1:i-1);
    U([i k],:) = U([k i],:);
    P([i k],:) = P([k i],:);
    % Da qui in poi si procede come in lu_nopiv.
    for j = i+1:n
      % Fatta la pivotazione, si può proseguire come
      % sopra.
      L(j,i) = U(j,i)/U(i,i);
      % Le righe successive all'i-esima sono ottenute
      % sottraendo un opportuno multiplo dell'i-esima.
      U(j,:) = U(j,:)-L(j,i)*U(i,:);
    end
```

```
  end
end
```

Esercizio 1.3. Scrivere una funzione che, presa in input una matrice reale A di ordine n non singolare, ne calcoli l'inversa usando la fattorizzazione LU.

In generale, l'inversa di A è una matrice $X = \begin{bmatrix} x_1 & \cdots & x_n \end{bmatrix}$ tale che

$$Ax_i = e_i \quad \text{per } i \in \{1, \dots, n\}.$$

Si tratta quindi di risolvere n sistemi lineari. Quando è disponibile la fattorizzazione LU di A, però si può seguire una strada che usa quanto implementato prima, ma che aumenta il numero di sistemi da risolvere. Sia quindi $PA = LU$, e quindi i sistemi diventano

$$LUx_i = Pe_i.$$

Le soluzioni x_i si determinano risolvendo i sistemi triangolari

$$Ly_i = Pe_i \quad \text{e} \quad Ux_i = y_i \quad \text{per } i \in \{1, \dots, n\}.$$

```
function [X] = inverse_lu(A)
  [L U P] = lu_piv(A);
  n = rows(A);
  for i=1:n
    Y = solve_lower_triangular(L, P(:,i));
    X(:,i) = solve_upper_triangular(U, Y);
  end
end
```

Esercizio 1.4. Scrivere una funzione che calcola il determinante di una matrice A usando la fattorizzazione $PA = LU$. Per calcolare il determinate di P, si modifichi la funzione scritta al punto 2 in modo da contare il numero di scambi di righe effettuati.

Da $PA = LU$ abbiamo che $\det P \det A = \det L \det U = \det U$. Essendo P una permutazione delle righe dell'identità, si ha $\det P = \pm 1$. Inoltre si nota che $PP = I$. Quindi, in principio, la funzione qui sotto potrebbe andare bene.

```
function x = lu_det1(A)
  [L U P] = lu_piv(A);
  x = det(P)*det(U);
end
```

Tuttavia questa implementazione richiede inutilmente alla macchina di calcolare L. Inoltre di P si sa che è I con le righe permutate, quindi $\det P$ è 1 se c'è stato un numero pari di scambi, -1 altrimenti. In breve, $\det P = (-1)^{\delta}$, dove δ è il numero di scambi di righe effettuati. Una versione più economica quindi è:

```
function [d] = lu_det2(A)
  n = rows(A); U = A;
  delta = 0;
  for i = 1:n
    m = 0; k = i;
    for j = i:n
      if abs(A(j,i)) > m
        k = j; m = abs(A(j,i));
```

```
      end
    end
    U([i k],:) = U([k i],:);
    if i != k
      delta = delta+1;
    end
    for j = i+1:n
      m = U(j,i)/U(i,i);
      U(j,:) = U(j,:)-m*U(i,:);
    end
  end
  % A questo punto U è una matrice triangolare: il suo
  % determinante è dato dal prodotto delle entrate
  % della diagonale principale.
  det_U = U(1,1);
  if n >= 2
    for i = 2:n
      det_U = det_U * U(i,i);
    end
  end
  d = (-1)^delta*det_U;
end
```

Esercizio 1.5. Scrivere due funzioni `sgs` e `mgs` che calcolino la fattorizzazione QR ridotta di una matrice, usando rispettivamente il metodo di Gram-Schmidt e il metodo di Gram-Schmidt modificato.

Siano $x_1, \dots, x_n \in \mathbb{R}^m$ linearmente indipendenti — quindi $m \geq n$ — e indichiamo con X la matrice che ha come colonne questi vettori. Allora, l'*algoritmo di Gram-Schmidt* dà n vettori $q_1, \dots, q_n \in \mathbb{R}^m$ tra loro ortogonali come segue:

$$q_1 := x_1$$
$$q_i := x_i - \sum_{j=1}^{i-1} \frac{q_j^T x_i}{\|q_j\|^2} q_j = x_i - \sum_{j=1}^{i-1} \left[\left(\frac{q_j}{\|q_j\|} \right)^T x_i \right] \frac{q_j}{\|q_j\|} \quad \text{per } i \in \{2, \dots, n\}$$

Se introduciamo la matrice R triangolare superiore attraverso

$$R_{j,i} := \begin{cases} \|q_i\| & \text{se } i = j \\ \left(\frac{q_j}{\|q_j\|} \right)^T x_i & \text{se } j \leq i-1 \\ 0 & \text{altrimenti} \end{cases}$$

possiamo scrivere

$$X = QR,$$

dove $Q := \left[\frac{q_1}{\|q_1\|} \quad \cdots \quad \frac{q_n}{\|q_n\|} \right]$.

Ecco l'implementazione di questo algoritmo.

```
function [Q R] = sgs(A)
  [m n] = size(A);
  Q = A;
  R = zeros(n, n);
  for i = 1:n
    for j = 1:i-1
      R(j,i) = Q(:,j)'*A(:,i);
      Q(:,i) = Q(:,i)-R(j,i)*Q(:,j);
```

```
    end
    R(i,i) = norm(Q(:,i));
    Q(:,i) = Q(:,i)/R(i,i);
  end
end
```

Per ragioni numeriche, è meglio però l'implementazione che segue:

```
function [Q R] = mgs(A)
  [m n] = size(A);
  R = zeros(n, n);
  for i = 1:n
    R(i,i) = norm(A(:,i));
    Q(:,i) = A(:,i)/R(i,i);
    for j = i+1:n
      R(i,j) = Q(:,i)'*A(:,j);
      A(:,j) = A(:,j)-R(i,j)*Q(:,i);
    end
  end
end
```

Esercizio 1.6. Sia $m = 50$ e $n = 12$. Sia $f(t) = \cos(4t)$ e si definiscano i punti $t_i = \frac{i-1}{m-1}$ per $i = 1, \dots, m$ e b il vettore di componenti $b_i = f(t_i)$. Sia A la matrice del problema dei minimi quadrati che si ottiene approssimando con un polinomio di grado $n-1$ la sequenza dei punti (t_i, b_i). Si risolva e visualizzi (con tutti e sedici i decimali) la soluzione del problema calcolata con i seguenti metodi:

1. formazione e soluzione del sistema di equazioni normali usando il comando \ di MATLAB
2. fattorizzazione QR utilizzando `sgs`
3. fattorizzazione QR utilizzando `mgs`
4. fattorizzazione QR utilizzando `qr` (funzione built-in di MATLAB che utilizza il metodo di Householder).

I calcoli precedenti generano quattro liste di 12 coefficienti. Prendendo come riferimento la soluzione con l'ultimo metodo, si cancellino in ogni lista le cifre decimali che appaiono errate (affette da arrori di arrotondamento). Quali metodi si mostrano più instabili? Non è necessario spiegare le osservazioni fatte.

In generale, presi m punti (x_1, y_1), ... e (x_m, y_m) di $\mathbb{R}^2$, si cerca un polinomio $p^* \in \mathbb{P}_n$, con $n < m-1$, tale che

$$\sum_{i=1}^{m} (p^*(x_i) - y_i)^2 \leq \sum_{i=1}^{m} (p(x_i) - y_i)^2 \quad \text{per ogni } p \in \mathbb{P}_n.$$

Per ricondurci al problema dei minimi quadrati, basta osservare che, se

$$p(x) = c_1 + c_2 x + c_3 x^3 + \dots + c_{n+1} x^n$$

allora

$$\sum_{i=1}^{m} (p(x_i) - y_i)^2 = \|Ac - y\|_2^2$$

dove

$$A := \begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_i & x_i^2 & \cdots & x_i^n \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_m & x_m^2 & \cdots & x_m^n \end{bmatrix} \in \mathbb{R}^{m\times(n+1)}.$$

Per trovare quindi il $p^* \in \mathbb{P}_n$ che minimizza $\sum_{i=0}^{n}(p(x_i) - y_i)^2$ è equivalente a trovare il $c \in \mathbb{R}^{n+1}$ che minimizza $\|Ac - y\|_2^2$. (Osserviamo a tal punto che A è invertibile.) Cioè si deve risolvere l'equazione

$$A^T Ac = A^T y$$

rispetto a c.

Diamo una funzione che presa una lista di ascisse $x \in \mathbb{R}^m$ e un numero naturale n produce la matrice A di sopra:

```
function [A] = vandermonde(n, x)
  % !!! Qui x è un vettore *colonna* di R^m
  for i = 0:n
    A(:,i+1) = x.^i;
  end
end
```

Diamo a anche le seguenti funzioni che risolvono il problema nei modi richiesti nell'ordine.

```
function [c] = minquad(n, x, y)
  % !!! Qui, x e y sono vettori *colonna* di R^m
  A = vandermonde(n, x');
  % Dobbiamo risolvere il problema A'*Ac = A'y in c.
  % Qui, lo facciamo con l'operatore \ di Matlab.
  c = (A'*A)\(A'*y);
end
```

Se usiamo la decomposizione $A = QR$ dell'esercizio precedente, allora

$$R^T \underbrace{Q^T Q}_{=I} Rc = R^T Q^T y$$

da cui, perché R è triangolare,

$$Rc = Q^T y.$$

Le funzioni che seguono si basano sulla risoluzione di questa equazione. Abbiamo già dato negli esercizi precedenti le funzioni che risolvono sistemi triangolari superiori.

```
function [c] = minquad_sgs(n, x, y)
  % !!! Qui, x e y sono vettori *colonna* di R^m
  A = vandermonde(n, x);
  % Dobbiamo risolvere il problema A'*Ac = A'y in c.
  [Q R] = sgs(A);
  c = solve_upper_triangular(R, Q'*y);
end
```

```
function [c] = minquad_mgs(n, x, y)
  % !!! Qui, x e y sono vettori *colonna* di R^m
  A = vandermonde(n, x');
```

```
  % Dobbiamo risolvere il problema A'*Ac = A'y in c.
  [Q R] = mgs(A);
  c = solve_upper_triangular(R, Q'*y);
end
```

```
function [c] = minquad_qr(n, x, y)
  A = vandermonde(n, x');
  [Q R] = qr(A);
  % Qui Q è una matrice m*m, mentre R è m*(n+1).
  % Ma R è trapeizoidale superiore, quindi
  Q = Q(:,1:n+1);
  R = R(1:n+1,:);
  c = (inverse_lu(R) * Q') * y;
end
```

Testiamo le funzioni appena implementate.

```
octave> format long;
octave> t = (@(i) (i-1)/49)(1:50)';
octave> b = (@(x) cos(4*x))(t);
octave> c1 = minquad(11, t, b)
warning: matrix singular to machine precision,
         rcond = 2.87439e-17
warning: called from
         minquad at line 6 column 5

warning: matrix singular to machine precision,
         rcond = 2.96518e-17
warning: called from
         minquad at line 6 column 5

c1 =

   9.999999998598327e-01
  -1.637969801790591e-07
  -7.999989743148679e+00
  -2.072197377938371e-04
   1.066866439686697e+01
  -1.069780529188339e-02
  -5.655104676152303e+00
  -6.193169964808305e-02
   1.679168741518508e+00
   1.575953725551536e-02
  -3.779623750302716e-01
   8.865738750316085e-02

octave> c2 = minquad_sgs(11, t, b)
c2 =

   1.000010858217738e+00
  -1.597441622478059e-03
  -7.961790735605360e+00
  -3.563609846140580e-01
   1.232237295280598e+01
  -4.198742023066941e+00
   1.143539844033863e-01
  -3.712350976184706e+00
   1.556815307995172e+00
```

```
   1.317363348231574e+00
  -7.841381000310229e-01
   5.040691898064982e-02

octave> c3 = minquad_mgs(11, t, b)
c3 =

   9.999999993467422e-01
   2.936344724349499e-08
  -7.999997687339519e+00
  -8.218757022590494e-05
   1.066765586639395e+01
  -5.947011460425414e-03
  -5.669068083912218e+00
  -3.556294129359597e-02
   1.647176449346650e+00
   3.985971904445812e-02
  -3.882198705483719e-01
   9.054209977801815e-02

octave> c4 = minquad_qr(11, t, b)
c4 =

   1.000000000996612e+00
  -4.227432701675582e-07
  -7.999981235680508e+00
  -3.187632773915539e-04
   1.066943079619887e+01
  -1.382028909574728e-02
  -5.647075624554418e+00
  -7.531602890230715e-02
   1.693606968736276e+00
   6.032104720361531e-03
  -3.742417021421716e-01
   8.040057586530689e-02
```

Prendendo come riferimento `c4`, possiamo vedere che `minquad_sgs` è meno affidabile, in quanto `c2` si discosta di molto a differenza degli altri risultati:

```
octave> norm(c4-c1)
ans = 2.390114932065619e-02
octave> norm(c4-c2)
ans = 8.288778082765266
octave> norm(c4-c3)
ans = 7.503944825237252e-02
```

2 *Metodi di ricerca di zeri*

Esercizio 2.1. Scrivere una funzione

```
[c, fc, iter] = bisezione(f, a, b, tol, itmax)
```

metodo di bisezione per la ricerca di una funzione `f` nell'intervallo `[a, b]`. Qui, `tol` è la tolleranza sull'approssimazione della radice e `itmax` è il numero massimo di iterazioni disposte a fare. Per quanto riguarda l'output, `c` è l'approssimazione trovata, `fc` è `f(c)` e `iter` è il numero di iterazioni fatte

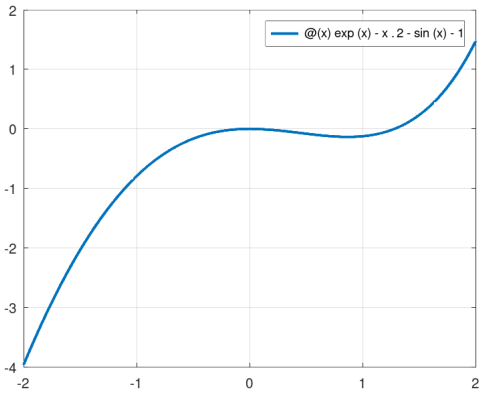

Figura 1. Grafico di $f(x) := e^x - x^2 - \sin x - 1$.

effettivamente per arrivare all'approssimazione c. Testare la funzione per la ricerca di una delle radici di

$$f(x) := e^x - x^2 - \sin x - 1$$

sull'intervallo $[-2, 2]$.

L'implementazione qui sotto usa il *teorema degli zeri*[1] e l'algoritmo di bisezione che descriviamo velocemente.

1. Iniziamo con $[a_0, b_0] := [a, b]$.
2. Per $n \in \mathbb{N}$, consideriamo $[a_n, b_n]$. Se f si annulla in uno tra i punti $a_n, b_n, c_n := \frac{a_n + b_n}{2}$, allora qualche zero è stato individuato. Altrimenti, si dà l'intervallo

$$[a_{n+1}, b_{n+1}] := \begin{cases} [a_n, c_n] & \text{se } f(a_n) f(c_n) < 0 \\ [c_n, b_n] & \text{se } f(c_n) f(b_n) > 0 \end{cases}.$$

A tal scopo è utile osservare che per ogni $c \in [a_n, b_n]$ si ha

$$|c - c_n| \leq \frac{b_n - a_n}{2} = \frac{b_0 - a_0}{2^{n+1}} \text{ per ogni } n \in \mathbb{N}.$$

Questa disuguaglianza dà un stima dell'errore commesso al passo n-esimo. Ovviamente, per un'implementazione al calcolatore non possiamo procedere indefinitamente: scegliamo di arrestarci quando viene superato un certo limite di iterazioni oppure quando il numero $\frac{b_n - a_n}{2}$ scende sotto un valore fissato.

```
function [c, fc, iter] = bisezione(f, a, b, tol, itmax)
  % INPUT:
  %   f     : funzione di cui cercare le radici
  %   a, b  : estremi delll'intervallo di ricerca
  %   tol   : sulla soluzione trovata
  %   itmax : numero massimo di iterazioni disposti
  %           a compiere per approssimare uno zero
  % OUTPUT:
  %   c     : approssimazione trovata
```

1 Sia $f : [a, b] \to \mathbb{R}$ continua con $f(a) f(b) < 0$. Allora esiste $c \in]a, b[$ tale che $f(c) = 0$.

```
  %   fc    : la funzione f valutata in c
  %   iter  : numero iterazioni effettuate
  iter = 0;
  while abs(a-b) > tol && iter < itmax
    % Nel caso in cui uno dei due estremi è
    % uno zero, ci fermiamo direttamente.
    if f(a) == 0
      c = a; return;
    elseif f(b) == 0
      c = b; return;
    else
      % Altrimenti, suddividiamo l'intervallo [a, b]
      % in due intervalli di uguale lunghezza, [a, c]
      % e [c, b], dove c è il punto medio di [a, c], e
      % si seleziona quello che ha uno zero.
      c = (a+b)/2;
      if f(c) == 0
        return;
      else
        if f(a)*f(c) < 0
          b = c; % lo zero sta nell'intervallo [a, c]
        else
          a = c; % lo zero sta nell'intervallo [c, b]
        end
      end
    end
  iter = iter+1;
  end
  fc = f(c);
end
```

Testiamo il metodo di bisezione su f:

```
octave> f = @(x) exp(x)-x.^2-sin(x)-1;
octave> [c, fc, iter] = bisezione(f, 1, 1.5, 1e-12, 100)
c = 1.279701331001888
fc = 6.681322162194192e-13
iter = 39
```

Esercizio 2.2. Scrivere una funzione

```
[c, fc, iter] = newton(f, df, x0, tol, itmax)
```

che implementa il metodo di Newton-Raphson per la ricerca di uno zero di una funzione `f` con derivata `df`. Qui, `x0` è il punto da cui parte il metodo, `tol` è la tolleranza sull'approssimazione della radice e `itmax` è il numero massimo di iterazioni disposte a fare. Per quanto riguarda l'output, `c` è l'approssimazione trovata, `fc` è `f(c)` e `iter` è il numero di iterazioni fatte effettivamente per arrivare all'approssimazione `c`. Testare la funzione per la ricerca di una delle radici di

$$f(x) := e^x - x^2 - \sin x - 1$$

sull'intervallo $[-2, 2]$.

Richiamiamo che il metodo di Newton-Raphson si scrive come

$$\begin{cases} x_0 \in \mathbb{R} \text{ scelto} \\ x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \text{ per } n \in \mathbb{N} \end{cases}$$

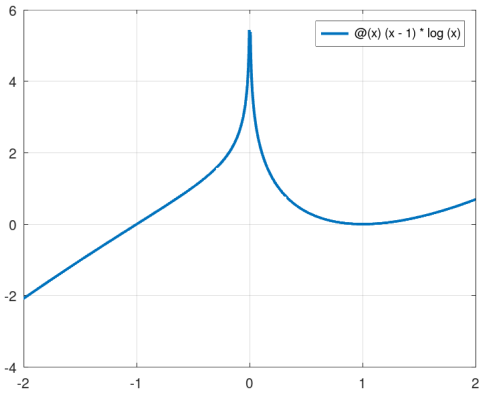

Figura 2. Grafico di $f(x) = (x-1)\ln x$.

Osserviamo che se dobbiamo controllare l'incremento, ci dobbiamo arrestare quando il valore assoluto di $\frac{f(x_n)}{f'(x_n)}$ scende al di sotto di una certa tolleranza. Anche in questo caso imponiamo un numero massimo di iterazioni.

```
function [a, fa, iter] = newton(f, df, a, tol, itmax)
  % INPUT:
  %   f     : funzione di cui cercare le radici
  %   df    : derivata prima della funzione f
  %   a     : punto da cui far partire il metodo
  %   tol   : sulla soluzione trovata
  %   itmax : numero massimo di iterazioni disposti
  %           a fare per l'approssimazione
  % OUTPUT:
  %   a     : approssimazione trovata
  %   fa    : la funzione f valutata in a
  %   iter  : numero massimo di iterazioni disposti
  %           a fare per arrivare ad a
  iter = 0;
  delta = tol+1; % per innescare il loop
  while iter < itmax && abs(delta) > tol
    iter = iter+1;
    delta = f(a)/df(a);
    a = a-delta;
  end
  fa = f(a);
end
```

Testiamo questo metodo su f:

```
octave> f = @(x) exp(x)-x.^2-sin(x)-1;
octave> df = @(x) exp(x)-2*x-cos(x);
octave> [c, fc, iter] = newton(f, df, 2, 1e-6, 100)
c = 1.279701331001091
fc = 7.083222897108499e-14
iter = 6
```

Esercizio 2.3. Considerare la funzione $f(x) = (x-1)\ln x$. Applicare il metodo di Newton-Raphson e il metodo di Newton-Raphson *modificato* per la ricerca

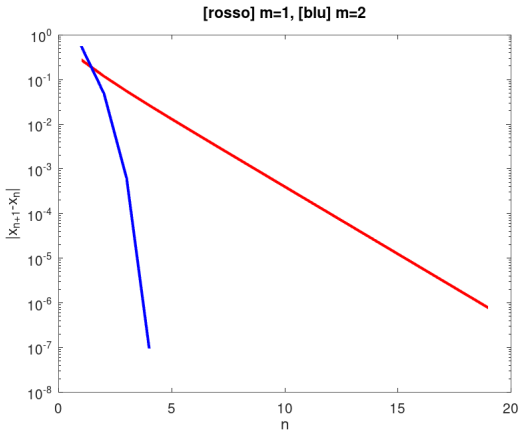

Figura 3. Grafico degli incrementi al variare del numero di iterazioni.

della radice doppia 1 e confrontare graficamente gli ordini di convergenza dei due metodi.

Se α è una radice con molteplicità $m \geq 2$, il metodo di Newton-Raphson ha ordine 1. Se però sappiamo a priori m, possiamo modificare questo metodo così:

$$\begin{cases} x_0 \in \mathbb{R} \text{ scelto} \\ x_{n+1} = x_n - m\frac{f(x_n)}{f'(x_n)} \text{ per } n \in \mathbb{N} \end{cases}$$

ed avere una convergenza quadratica.

Modifichiamo `newton` in questo modo: tra gli output teniamo conto di m, c'è un solo output questa volta ed è una lista degli incrementi $|x_{n+1} - x_n|$.

```
function [errs] = newton_errors(f, df, a, m, tol, itmax)
  % INPUT:
  %   f     : funzione di cui cercare le radici
  %   df    : derivata prima della funzione f
  %   a     : punto da cui far partire il metodo
  %   m     : molteplicità della radice attesa
  %   tol   : sulla soluzione trovata
  %   itmax : numero massimo di iterazioni disposti
  %           a fare per l'approssimazione
  % OUTPUT:
  %   errs  : lista degli errori commessi da un passo
  %           al successivo
  iter = 0;
  delta = tol+1; % per innescare il loop
  while iter < itmax && abs(delta) > tol
    iter = iter+1;
    delta = m * f(a)/df(a);
    a = a-delta;
    errs(iter) = abs(delta);
  end
end
```

Il seguente script disegna il grafico che permette di confrontare la velocità di convergenza nei casi `m=1` e `m=2`.

```
format long;
```

```
2
3  % Dati del problema.
4  f = @(x) (x-1)*log(x);
5  df = @(x) log(x) + (x-1)/x;
6  x0 = 1.5;
7  tol = 1e-6;
8  itmax = 100;
9
10 % Grafico della funzione.
11 fplot(f, [-2, 2], "LineWidth", 2);
12 grid on;
13 print graph2.png
14
15 % Collezione degli incrementi e relativi grafici.
16 [errs1] = newton_errors(f, df, x0, 1, tol, itmax);
17 [errs2] = newton_errors(f, df, x0, 2, tol, itmax);
18 semilogy(errs1, "red", "LineWidth", 2);
19 hold on;
20 semilogy(errs2, "blue", "LineWidth", 2);
21 hold off;
22 title(["[rosso] m=1, " "[blu] m=2"]);
23 xlabel("n");
24 ylabel("|x_{n+1}-x_n|");
25 print convergences.png
```