

# Analisi Numerica

## Relazioni di Laboratorio

Indrjo Dedej

Ultima revisione: 26 maggio 2023.

### Indice

1	Metodi diretti per sistemi lineari	1
2	Metodi di ricerca di zeri	9

**Nota 0.1.** Per la parte dei codici è stato usato GNU Octave al posto 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  $x := [x_1 \ \dots \ x_n]^T$  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  $Ax = b$ .

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

## 1. Metodi diretti per sistemi lineari

```

4   for i = 2:n
5       x(i) = (b(i) - A(i,1:i-1) * x(1:i-1))/A(i,i);
6   end
7   x = x';
8 end

```

Invece, la soluzione  $x := [x_1 \ \cdots \ x_n]^T$  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

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

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

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

```

1 function [x] = solve_upper_triangular(A,b)
2     n = length(b);
3     x(n) = b(n)/A(n,n);
4     for i = 1:n
5         i = n-i+1;
6         x(i) = (b(i) - A(i,i+1:n) * x(i+1:n))/A(i,i);
7     end
8     x = x';
9 end

```

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

```

1 function [x] = solve_triangular(A,b)
2     if istril(A)
3         [x] = solve_lower_triangular(A,b);
4     elseif istriu(A)
5         [x] = solve_upper_triangular(A,b);
6     else
7         disp("the matrix isn't triangular!");
8         return;
9     end
10 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.

```

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

```

La fattorizzazione LU con pivotazione coinvolge un'altra matrice:

$$PA = LU$$

dove  $P$  è ottenuta dall'identità permutandone le righe.

```

1 function [L U P] = lu_piv(A)
2     n = rows(A);
3     L = eye(n);
4     U = A;
5     P = eye(n);
6     for i = 1:n
7         % Ricerca del primo j per cui |A(j,i)| è massimo
8         m = 0; k = i;
9         for j = i:n
10             if abs(A(j,i)) > m
11                 k = j; m = abs(A(j,i));
12             end
13         end
14         L([i k],1:i-1) = L([k i],1:i-1);
15         U([i k],:) = U([k i],:);
16         P([i k],:) = P([k i],:);
17         % Da qui in poi si procede come in lu_nopiv.
18         for j = i+1:n
19             % Fatta la pivotazione, si può proseguire come
20             % sopra.
21             L(j,i) = U(j,i)/U(i,i);
22             % Le righe successive all'i-esima sono ottenute
23             % sottraendo un opportuno multiplo dell'i-esima.
24             U(j,:) = U(j,:) - L(j,i)*U(i,:);
25         end
26     end
27 end

```

```

26 end
27 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  $X = [x_1 \ \cdots \ x_n]$  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

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

```

1 function [X] = inverse_lu(A)
2   [L U P] = lu_piv(A);
3   n = rows(A);
4   for i=1:n
5     Y = solve_lower_triangular(L, P(:,i));
6     X(:,i) = solve_upper_triangular(U, Y);
7   end
8 end

```

**Esercizio 1.4.** Scrivere una funzione che calcola il determinante di una matrice  $A$  usando la fattorizzazione  $PA = LU$ . Per calcolare il determinante 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.

```

1 function x = lu_det1(A)
2   [L U P] = lu_piv(A);
3   x = det(P)*det(U);
4 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  $\delta$  è il numero di scambi di righe effettuati. Una versione più economica quindi è:

```

1 function [d] = lu_det2(A)
2   n = rows(A); U = A;
3   delta = 0;
4   for i = 1:n
5     m = 0; k = i;
6     for j = i:n
7       if abs(A(j,i)) > m
8         k = j; m = abs(A(j,i));

```

```

9     end
10    end
11    U([i k], :) = U([k i], :);
12    if i != k
13        delta = delta+1;
14    end
15    for j = i+1:n
16        m = U(j,i)/U(i,i);
17        U(j,:) = U(j,:)-m*U(i,:);
18    end
19 end
20 % A questo punto U è una matrice triangolare: il suo
21 % determinante è dato dal prodotto delle entrate
22 % della diagonale principale.
23 det_U = U(1,1);
24 if n >= 2
25     for i = 2:n
26         det_U = det_U * U(i,i);
27     end
28 end
29 d = (-1)^delta*det_U;
30 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  $a_1, \dots, a_n \in \mathbb{R}^m$  linearmente indipendenti. L'algoritmo di Gram-Schmidt dà  $n$  vettori  $q_1, \dots, q_n \in \mathbb{R}^m$  tra loro ortogonali:

$$q_1 := x_1$$

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

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

Ecco l'implementazione di questo algoritmo.

```

1 function [Q R] = sgs(A)
2     [m n] = size(A);
3     Q = A;
4     R = zeros(n, n);
5     for i = 1:n
6         for j = 1:i-1
7             R(j,i) = Q(:,j)'*A(:,i);
8             Q(:,i) = Q(:,i)-R(j,i)*Q(:,j);
9         end

```

```

10 R(i,i) = norm(Q(:,i));
11 Q(:,i) = Q(:,i)/R(i,i);
12 end
13 end

```

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

```

1 function [Q R] = mgs(A)
2 [m n] = size(A);
3 R = zeros(n, n);
4 for i = 1:n
5     R(i,i) = norm(A(:,i));
6     Q(:,i) = A(:,i)/R(i,i);
7     for j = i+1:n
8         R(i,j) = Q(:,i)'*A(:,j);
9         A(:,j) = A(:,j)-R(i,j)*Q(:,i);
10    end
11 end
12 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 errori di arrotondamento). Quali metodi si mostrano più instabili? Non è necessario spiegare le osservazioni fatte.

In generale, presi  $m$  punti  $(x_1, y_1), \dots, (x_m, y_m)$  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

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

## 1. Metodi diretti per sistemi lineari

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  $\|Ac - y\|_2^2$ . Cioè si deve risolvere l'equazione

$$A^T A c = A^T y$$

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:

```
1 function [A] = vandermonde(n, x)
2 % !!! Qui x è un vettore *colonna* di R^m
3 for i = 0:n
4     A(:,i+1) = x.^i;
5 end
6 end
```

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

```
1 function [c] = minquad(n, x, y)
2 % !!! Qui, x e y sono vettori *colonna* di R^m
3 A = vandermonde(n, x');
4 % Dobbiamo risolvere il problema A'*Ac = A'y in c.
5 % Qui, lo facciamo con l'operatore \ di Matlab.
6 c = (A'*A) \ (A'*y);
7 end
```

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

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

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.

```
1 function [c] = minquad_sgs(n, x, y)
2 % !!! Qui, x e y sono vettori *colonna* di R^m
3 A = vandermonde(n, x);
4 % Dobbiamo risolvere il problema A'*Ac = A'y in c.
5 [Q R] = sgs(A);
6 c = solve_upper_triangular(R, Q'*y);
7 end
```

```
1 function [c] = minquad_mgs(n, x, y)
2 % !!! Qui, x e y sono vettori *colonna* di R^m
3 A = vandermonde(n, x');
4 % Dobbiamo risolvere il problema A'*Ac = A'y in c.
5 [Q R] = mgs(A);
6 c = solve_upper_triangular(R, Q'*y);
7 end
```

```
1 function [c] = minquad_qr(n, x, y)
2 A = vandermonde(n, x');
3 % Dobbiamo risolvere il problema A'*Ac = A'y in c.
```

## 1. Metodi diretti per sistemi lineari

```

4 [Q R] = sgs(A);
5 % Qui Q è una matrice m*m, mentre R è m*(n+1).
6 % Ma R è trapezoidale superiore, quindi
7 Q = Q(:,1:n+1);
8 R = R(1:n+1,:);
9 c = (inverse_lu(R) * Q') * y;
10 end

```

Testiamo le funzioni appena implementate. Prima di tutto, diamo i vettori di  $\mathbb{R}^{50}$ :

```

octave> format long
octave> t = (@(i) (i-1)/49)(1:50)';
octave> b = (@(x) cos(4*x))(t);

```

Chiediamo ora di risolvere il problema ai minimi quadrati, usando un polinomi di grado 11:

```

octave> c1 = minquad(11, t, b)
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 =

    1.000010858217736e+00
   -1.597441622479323e-03
   -7.961790735605575e+00
   -3.563609846100917e-01
    1.232237295279629e+01
   -4.198742022992519e+00
    1.143539843615144e-01
   -3.712350976158632e+00
    1.556815307967554e+00
    1.317363348240178e+00
   -7.841381000343972e-01
    5.040691898039995e-02

```

```

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

    9.999999993467416e-01
    2.936344856152573e-08
   -7.999997687339061e+00
   -8.218757375288988e-05
    1.066765586643305e+01
   -5.947011668467894e-03

```



## 2. Metodi di ricerca di zeri

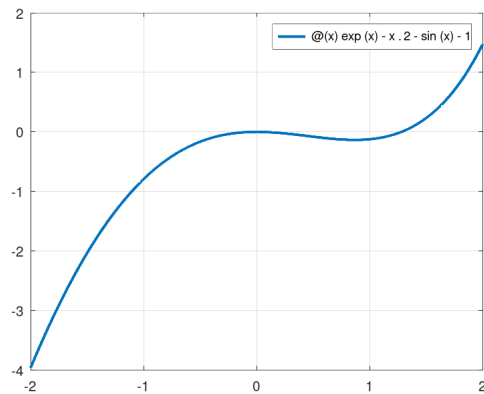


Figura 1. Grafico di  $f(x) := e^x - x^2 - \sin x - 1$ .

```
-5.669068083108868e+00
-3.556294157169759e-02
 1.647176449303515e+00
 3.985971841029823e-02
-3.882198705396149e-01
 9.054209982423345e-02

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

 1.000000000996612e+00
-4.227432497394545e-07
-7.999981235680764e+00
-3.187632846675115e-04
 1.066943079626799e+01
-1.382028918305878e-02
-5.647075624787249e+00
-7.531602913513780e-02
 1.693606968736276e+00
 6.032105768099427e-03
-3.742417020548601e-01
 8.804057586530689e-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,  $\text{tol}$  è la tolleranza sull'approssimazione della radice e  $\text{itmax}$  è il numero massimo di iterazioni disposte a fare. Per quanto riguarda l'output,  $c$  è l'approssimazione trovata,  $fc$  è  $f(c)$  e  $\text{iter}$  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  $[-2, 2]$ .

L'implementazione qui sotto usa il *teorema degli zeri*:

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

e l'algoritmo di bisezione che descriviamo velocemente.

1.  $[a_0, b_0] := [a, b]$ .
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

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

Osserviamo che per ogni  $c \in [a_n, b_n]$  si ha

$$|c - c_n| \leq \frac{a_n + b_n}{2} \text{ per ogni } n \in \mathbb{N}$$

e che quindi

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

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

$$\frac{a_n - b_n}{2}$$

scende sotto una fissato valore.

```

1 function [c, fc, iter] = bisezione(f, a, b, tol, itmax)
2   % INPUT:
3   %   f      : funzione di cui cercare le radici
4   %   a, b    : estremi dell'intervallo di ricerca
5   %   tol     : sulla soluzione trovata
6   %   itmax   : numero massimo di iterazioni disposti
7   %             a compiere per approssimare uno zero
8   % OUTPUT:
9   %   c       : approssimazione trovata
10  %   fc      : la funzione f valutata in c
11  %   iter     : numero iterazioni effettuate
12  iter = 0;
13  while abs(a-b)/2 > tol && iter < itmax
14    % Nel caso in cui uno dei due estremi è
15    % uno zero, ci fermiamo direttamente.
16    if f(a) == 0
17      c = a; return;
18    elseif f(b) == 0
19      c = b; return;
20    else
21      % Altrimenti, suddividiamo l'intervallo [a, b]
22      % in due intervalli di uguale lunghezza, [a, c]
23      % e [c, b], dove c è il punto medio di [a, b], e
24      % si seleziona quello che ha uno zero.
25      c = (a+b)/2;
26      if f(c) == 0

```

## 2. Metodi di ricerca di zeri

```

27     return;
28   else
29     if f(a)*f(c) < 0
30       b = c; % lo zero sta nell'intervallo [a, c]
31     else
32       a = c; % lo zero sta nell'intervallo [c, b]
33     end
34   end
35 end
36 iter = iter+1;
37 end
38 fc = f(c);
39 end

```

Testiamo il metodo di bisezione su  $f$ :

```

>> f = @(x) exp(x)-x.^2-sin(x)-1;
>> [c, fc, iter] = bisezione(f, 1, 1.5, 1e-12, 100)
c = 1.279701331002798
fc = 1.349587108734340e-12
iter = 38

```

**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,  $x_0$  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

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

Osserviamo che se dobbiamo controllare l'incremento, ci dobbiamo arrestare appena che il valore assoluto di  $\frac{f(x_n)}{f'(x_n)}$  scende al di sotto di una certa tolleranza.

```

1 function [a, fa, iter] = newton(f, df, a, tol, itmax)
2   % INPUT:
3   % f      : funzione di cui cercare le radici
4   % df     : derivata prima della funzione f
5   % a      : punto da cui far partire il metodo
6   % tol    : sulla soluzione trovata
7   % itmax  : numero massimo di iterazioni disposti
8   %        : a fare per l'approssimazione
9   % OUTPUT:
10  % a      : approssimazione trovata
11  % fa     : la funzione f valutata in a
12  % iter   : numero massimo di iterazioni disposti
13  %        : a fare per arrivare ad a

```

## 2. Metodi di ricerca di zeri

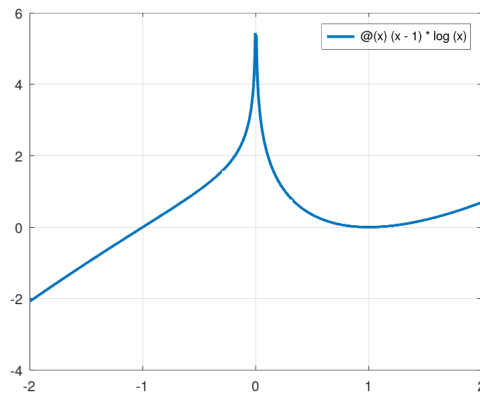


Figura 2. Grafico di  $f(x) = (x-1)\ln x$ .

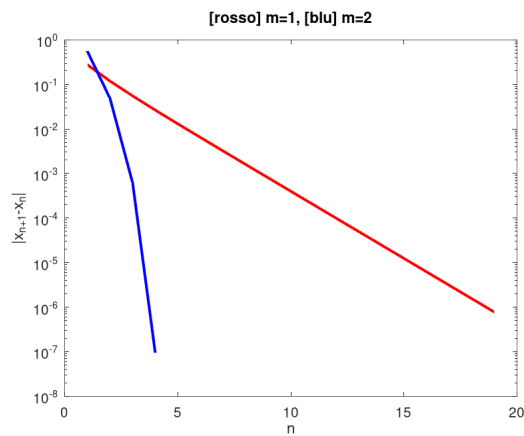


Figura 3. Grafico degli incrementi al variare del numero di iterazioni.

```

14 iter = 0;
15 delta = tol+1; % per innescare il loop
16 while iter < itmax && abs(delta) > tol
17     iter = iter+1;
18     delta = f(a)/df(a);
19     a = a-delta;
20 end
21 fa = f(a);
22 end

```

Testiamo questo metodo su  $f$ :

```

>> f = @(x) exp(x)-x.^2-sin(x)-1;
>> df = @(x) exp(x)-2*x-cos(x);
>> [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

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

Se  $\alpha$  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  $|x_{n+1} - x_n|$ .

```

1 function [errs] = newton_errors(f, df, a, m, tol, itmax)
2     % INPUT:
3     % f      : funzione di cui cercare le radici
4     % df     : derivata prima della funzione f
5     % a      : punto da cui far partire il metodo
6     % m      : molteplicità della radice attesa
7     % tol    : sulla soluzione trovata
8     % itmax  : numero massimo di iterazioni disposti
9     %          a fare per l'approssimazione
10    % OUTPUT:
11    % errs   : lista degli errori commessi da un passo
12    %          al successivo
13    iter = 0;
14    delta = tol+1; % per innescare il loop
15    while iter < itmax && abs(delta) > tol
16        iter = iter+1;
17        delta = m * f(a)/df(a);
18        a = a-delta;
19        errs(iter) = abs(delta);
20    end
21 end

```

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

```

1 format long;
2
3 f = @(x) (x-1)*log(x);
4 df = @(x) log(x) + (x-1)/x;
5 x0 = 1.5;
6 tol = 1e-6;
7 itmax = 100;
8
9 fplot(f, [-2,2], "LineWidth", 2); grid on;
10 print graph2.png
11
12 [errs1] = newton_errors(f, df, x0, 1, tol, itmax);
13 [errs2] = newton_errors(f, df, x0, 2, tol, itmax);
14
15 semilogy(errs1, "red", "LineWidth", 2);
16 hold on;
17 semilogy(errs2, "blue", "LineWidth", 2);
18 hold off;
19 title(["[rosso] m=1, " "[blu] m=2"]);
20 xlabel("n");

```

```
21 | ylabel("|x_{n+1}-x_n|");  
22 | print convergences.png
```