```
----------- radqua_N.m ------------------------
function r= radqua_N(x,N)
% Calcola N valori della successione
% r_0 = x
% r_{k+1} = (r_k + x / r_k) / 2
% e restituisce l'ultimo valore calcolato

r=x;
for k=1:N
   r=0.5*(r+x/r);
end
----------------------------------------
>> r=radqua_N(3,10)
r =
    1.7321
>> r-sqrt(3)
ans =
     0
>> r=radqua_N(357,10)
r =
   18.8944
>> r-sqrt(357)
ans =
     0
>> r=radqua_N(30057,10)
r =
  173.3721
>> r-sqrt(30057)
ans =
    0.0026
% La precisione è molto bassa
>> r=radqua_N(30057,11)
r =
  173.3695
>> r-sqrt(30057)
ans =
   1.9025e-08
% Aumentando le iterazioni, abbiamo un errore pìu piccolo

% Programmiamo un metodo con stima d'errore
```

```
--------- radqua1.m ------------------------------------
function [r,N] = radqua1(x,toll)
% Approssima sqrt(x) a meno di toll
% usando la successione
% r_0 = x
% r_{k+1} = (r_k + x / r_k) / 2
% e la stima d'errore
% e_{k+1} <= L^k / (1-L) * |r1-r0|
% con L=0.5-0.5*x/r0^2

% Se x<1, il metodo converge, ma la stima d'errore non
% è più valida, quindi in questo caso invertiamo il numero
if (x<1)
   x=1/x;
   flag=true;
else
   flag=false;
end

r=x; N=0;
r1=0.5*(r+x/r);
D=abs(r1-r);
L=0.5-0.5*x/r^2;
while (L^N/(1-L)*D > toll)
   r=0.5*(r+x/r);
   N = N+1;
end
% Se avevamo invertito x, invertiamo la radice
if flag
   r=1/r;
end
-----------------------------------------------
>> [r,N]=radqua1(1597,0.1)
r =
  39.9625
N =
   14
>> r-sqrt(1597)
ans =
    0
```

```
>> [r,N]=radqua1(0.3,0.1)
r =
   0.5477
N =
    3
>> r-sqrt(.3)
ans =
  -5.8245e-05

%Osserviamo che la stima d'errore funziona (il risultato è sempre
% calcolato a meno della tolleranza richiesta), ma l'errore vero
% è molto più piccolo di quello stimato e quindi probabilmente il
% metodo radqua1.m impega un numero eccessivamente alto di iterazioni

% Proviamo con una stima d'errore più precisa

-------- radqua.m -----------------------------------------
function [r,n]=radqua(x,toll)
% Approssima sqrt(x) a meno di toll
% usando la successione
% r_0 = x
% r_{k+1} = (r_k + x / r_k) / 2
% e la stima d'errore
% e_{k+1} <= L^k / (1-L) * |r_{k+1}-r_{k}|
% con L=0.5-0.5*x/r_k^2

if nargin<2
   toll=1e-6;
end
if (x<0)
   error('Il radicando deve essere positivo');
end

% Vogliamo che la succ sia monotona descrescente
% in modo da poter stimare L dall'alto.
if x<1
   flag=true;
   x=1/x;
else
   flag = false;
```

```
end
err=1+toll;
r=x;
n=0;
while (err>toll)
   r1=0.5*(r+x/r);
   L=0.5*(1-x/r^2);
   err=abs(r1-r)*L/(1-L);
   r=r1;
   n=n+1;
end
if flag
   r=1/r;
end
----------------------------------------
% In questa versione, man mano che il calcolo procede, se r0>1,
% sia L che (r1-r) decrescono e la stima d'errore diventa via
% via più stringente. Confrontiamo i metodi:


>> [r,N]=radqua1(1597,0.01)
r =
  39.9625
N =
   18
>> r-sqrt(1597)
ans =
    0
>> [r,N]=radqua(1597,0.01)
r =
  39.9627
N =
    8
>> r-sqrt(1597)
ans =
  2.1744e-04

% La nuova implementazione è migliore perché usa un numero
% di iterazioni più rispondente alla precisione richiesta
% dall'utente.
```

```
>> [r,N]=radqua(15970000,0.01)
r =
  3.9962e+03
N =
   15
>> [r,N]=radqua(1597000000,0.01)
r =
  3.9962e+04
N =
   19
>> [r,N]=radqua(159700000000,0.01)
r =
  3.9962e+05
N =
   22

% Osserviamo che il numero di iterazioni necessarie aumenta
%all'aumentare  del radicando e quindi creiamo una nuova versione
%dell'algoritmo che divida il radicando x per 4^m, calcoli la radice
%di (x/4^m) col metodo iterativo e poi moltiplichi il risultato per
%2^m. Questo modo di procedere è conveniente se posso trovare
%facilmente un esponente m tale che x/4^m stia (ad esempio) in [1,4]
%Questo problema è facilmente risolubile usando la rappresentazione
%binaria del numero floating point x (funzione log2 in MatLab, logb()
%in C/C++) e usando la moltiplicazione veloce per potenze di 2 che
%modifica solo l'esponente della rappresentazione binaria (funzione
%pow2 in MatLab e ldexp() in C/C++). Si veda 'doc log2' in MatLab.

------------ radqua_fast ---------------------------------------
function [r,N] = radqua_fast(x,toll)
% Approssima sqrt(x) a meno di toll
% usando la successione
% r_0 = x
% r_{k+1} = (r_k + x / r_k) / 2
% e la stima d'errore
% e_{k+1} <= L^k / (1-L) * |r_{k+1}-r_{k}|
% con L=0.5-0.5*x/r_k^2
%Questa versione è pensata per un dispositivo
% dalle risorse limitate su cui possiamo facilmente accedere
```

```
% all'esponente della rappresentazione floating point...

if (x<0)
    error('Il radicando deve essere positivo');
end

[f,e]=log2(x);         %mantissa ed esponente di x
m = floor( (e-1)/2 ); %scelgo la potenza di 4
x = pow2(x,-2*m);      % divide x per 2^(2m)=4^m
% ora x sta fra 1 e 4

r=x; N=0; err = 2*toll;
while (err > toll)
    r1=0.5*(r+x/r);
    L=0.5*(1-x/r^2);
    err = L/(1-L)*(r1-r); %abs si può togliere, tanto r1>r se r0>1
    N = N+1;
    r = r1;
end
r = pow2(r,m); %moltiplicazione per 2^m
-------------------------------------------------------------

>> [r,N]=radqua(159700000000,0.01)
r =
   3.9962e+05
N =
    22
>> [r,N]=radqua_fast(159700000000,0.01)
r =
   3.9963e+05
N =
     3
>> [r,N]=radqua(0.00000000000356,0.01)
r =
   1.8868e-06
N =
    23
>> [r,N]=radqua_fast(0.00000000000356,0.01)
r =
   1.8873e-06
```

```
N =
    3

% Ora il numero di iterazioni dipende meno fortemente da x ed è
% contenuto anche per valori di x molto grandi (o molto piccoli)

%Osservazione: ora sqrt(x/4^m) è calcolata a meno di toll e quindi,
% per la propagazione degli errori, abbiamo che sqrt(x) è calcolata a
% meno di toll*2^m, ma questo significa soltanto che il valore di toll
% che abbiamo fornito in ingresso è interpretato come tolleranza
% relativa (e non tolleranza assoluta come in radqua.m)

>> for x=logspace(-3,8,100)
[r,N]=radqua_fast(x,1e-3);
disp(sprintf('radice di %4.3e = %4.3e, calcolata in %d iterazioni',x,r,N))
end

radice di 1.000e-03 = 3.163e-02, calcolata in 1 iterazioni
radice di 1.292e-03 = 3.594e-02, calcolata in 2 iterazioni
radice di 1.668e-03 = 4.084e-02, calcolata in 3 iterazioni
radice di 2.154e-03 = 4.642e-02, calcolata in 3 iterazioni
radice di 2.783e-03 = 5.275e-02, calcolata in 3 iterazioni
radice di 3.594e-03 = 5.996e-02, calcolata in 3 iterazioni
radice di 4.642e-03 = 6.813e-02, calcolata in 2 iterazioni
radice di 5.995e-03 = 7.745e-02, calcolata in 2 iterazioni
radice di 7.743e-03 = 8.799e-02, calcolata in 3 iterazioni
....
radice di 1.292e+07 = 3.594e+03, calcolata in 3 iterazioni
radice di 1.668e+07 = 4.084e+03, calcolata in 4 iterazioni
radice di 2.154e+07 = 4.642e+03, calcolata in 2 iterazioni
radice di 2.783e+07 = 5.275e+03, calcolata in 3 iterazioni
radice di 3.594e+07 = 5.995e+03, calcolata in 3 iterazioni
radice di 4.642e+07 = 6.813e+03, calcolata in 3 iterazioni
radice di 5.995e+07 = 7.744e+03, calcolata in 3 iterazioni
radice di 7.743e+07 = 8.799e+03, calcolata in 2 iterazioni
radice di 1.000e+08 = 1.000e+04, calcolata in 2 iterazioni

>> for x=logspace(-3,8,100)
[r,N]=radqua_fast(x,1e-8);
disp(sprintf('radice di %4.3e = %4.3e, calcolata in %d iterazioni',x,r,N))
```

```
end
radice di 1.000e-03 = 3.162e-02, calcolata in 2 iterazioni
radice di 1.292e-03 = 3.594e-02, calcolata in 3 iterazioni
radice di 1.668e-03 = 4.084e-02, calcolata in 4 iterazioni
radice di 2.154e-03 = 4.642e-02, calcolata in 4 iterazioni
radice di 2.783e-03 = 5.275e-02, calcolata in 4 iterazioni
radice di 3.594e-03 = 5.995e-02, calcolata in 5 iterazioni
radice di 4.642e-03 = 6.813e-02, calcolata in 3 iterazioni
....
radice di 1.668e+07 = 4.084e+03, calcolata in 5 iterazioni
radice di 2.154e+07 = 4.642e+03, calcolata in 3 iterazioni
radice di 2.783e+07 = 5.275e+03, calcolata in 4 iterazioni
radice di 3.594e+07 = 5.995e+03, calcolata in 4 iterazioni
radice di 4.642e+07 = 6.813e+03, calcolata in 4 iterazioni
radice di 5.995e+07 = 7.743e+03, calcolata in 5 iterazioni
radice di 7.743e+07 = 8.799e+03, calcolata in 3 iterazioni
radice di 1.000e+08 = 1.000e+04, calcolata in 4 iterazioni

% Pensando a come è fatto l'algoritmo, il numero massimo iterazioni
%necessarie per ottenere una data precisione sarà necessario quando
%x/4^m risulta vicino a 4, quindi lo possiamo stimare sperimentalmente
%così

>> [r,N]=radqua_fast(3.99999999999,1e-8)
r =
   2.0000
N =
    5
>> [r,N]=radqua_fast(3.99999999999,1e-3)
r =
   2.0000
N =
    4

% Volendo, per un'applicazione specifica in cui serva sempre la stessa
%precisione, potremmo addirittura togliere i controlli con la stima
%dell'errore e far eseguire un numero fissato di iterazioni che
%abbiamo precedentemente stimato come indicato qui sopra... (Se lo
%fate, lasciate un commento ben chiaro nel codice!!!!!)
```