

Matlab: complessità e stabilità degli algoritmi. Alcuni esempi.

Ángeles Martínez Calomardo e Alvisé Sommariva

Università degli Studi di Padova

29 ottobre 2012

Stabilità : radici secondo grado

Dato $x^2 + 2px - q$, con $\sqrt{p^2 + q} \geq 0$ eseguiamo un primo algoritmo Matlab che valuta la radice via:

$$y = -p + \sqrt{p^2 + q}. \quad (1)$$

- ▶ $\sqrt{p^2 + q} \geq 0$ implica radici reali.
- ▶ Potenzialmente instabile per $p \gg q$ a causa della sottrazione tra p e $\sqrt{p^2 + q}$ (cancellazione).

Valutiamo radice con un secondo algoritmo stabile via razionalizzazione di (1):

$$\begin{aligned} y &= -p + \sqrt{p^2 + q} = \frac{(-p + \sqrt{p^2 + q})(p + \sqrt{p^2 + q})}{(p + \sqrt{p^2 + q})} \\ &= \frac{q}{(p + \sqrt{p^2 + q})} \end{aligned} \quad (2)$$

Codice stabilità : algoritmo 1

Salviamo il seguente codice in `radicesecgrado.m`.

```
p=1000; q=0.0180000000081; sol=0.9*10^(-5);  
  
% ALGORITMO 1  
s=p^2;  
t=s+q;  
if t >=0  
    u=sqrt(t);  
else  
    fprintf('\n \t [RADICI COMPLESSE] ');  
end  
s1=-p+u;
```

Codice stabilità : algoritmo 2

```
% ALGORITMO 2
s=p^2;
t=s+q;
if t >=0
    u=sqrt(t);
else
    fprintf('\n \t [RADICI COMPLESSE] ');
end
v=p+u;
t1=q/v;
```

Codice stabilità : stampa risultati

```
fprintf('\n \t [ALG.1]: %10.19f',s1);  
fprintf('\n \t [ALG.2]: %10.19f',t1);  
if length(sol) > 0 & (sol ~= 0)  
    rerr1 =abs(s1-sol)/abs(sol);  
    rerr2=abs(t1-sol)/abs(sol);  
    fprintf('\n \t [REL.ERR.ALG.1]: %2.2e',rerr1);  
    fprintf('\n \t [REL.ERR.ALG.2]: %2.2e',rerr2);  
end
```

Come previsto, il secondo algoritmo si comporta notevolmente meglio del primo, che compie un errore relativo dell'ordine di circa 10^{-9} . Infatti:

```
>> radicesecgrado  
  
[ALG.1] [1]: 0.0000089999999772772  
[ALG.2] [1]: 0.0000090000000000000  
[REL.ERR.][ALG.1]: 2.52e-009  
[REL.ERR.][ALG.2]: 0.00e+000  
  
>>
```

Eseguiamo un codice Matlab che valuti le successioni $\{u_n\}$, $\{z_n\}$, definite rispettivamente come

$$\begin{cases} s_1 = 1, & s_2 = 1 + \frac{1}{4} \\ u_1 = 1, & u_2 = 1 + \frac{1}{4} \\ s_{n+1} = s_n + \frac{1}{(n+1)^2} \\ u_{n+1} = \sqrt{6 s_{n+1}} \end{cases}$$

e

$$\begin{cases} z_1 = 1, & z_2 = 2 \\ z_{n+1} = 2^{n-\frac{1}{2}} \sqrt{1 - \sqrt{1 - 4^{1-n} \cdot z_n^2}} \end{cases} \quad (3)$$

che *teoricamente* convergono a π .

Implementiamo poi la successione, diciamo $\{y_n\}$, che si ottiene *razionalizzando* (3), cioè moltiplicando numeratore e denominatore di

$$z_{n+1} = 2^{n-\frac{1}{2}} \sqrt{1 - \sqrt{1 - 4^{1-n} \cdot z_n^2}}$$

per

$$\sqrt{1 + \sqrt{1 - 4^{1-n} \cdot z_n^2}}$$

e calcoliamo u_m , z_m e y_m per $m = 2, 3, \dots, 40$ (che teoricamente dovrebbero approssimare π).

Infine disegniamo in un unico grafico l'andamento dell'errore relativo di u_n , z_n e y_n rispetto a π aiutandoci con l'help di Matlab relativo al comando `semilogy`.

Calcolo π : metodo 1

In seguito scriviamo un'implementazione di quanto richiesto commentando i risultati. Si salvi in un file `pigreco.m` il codice

```
% SEQUENZE CONVERGENTI "PI GRECO".
```

```
% METODO 1.
```

```
s(1)=1; u(1)=1;
```

```
s(2)=1.25; u(2)=s(2);
```

```
for n=2:40
```

```
    s(n+1)=s(n)+(n+1)^(-2);
```

```
    u(n+1)=sqrt(6*s(n+1));
```

```
end
```

```
rel_err_u=abs(u-pi)/pi;
```

```
fprintf('\n');
```

Calcolo π : metodo 2

```
% METODO 2.  
format long  
z(1)=1;  
z(2)=2;  
for n=2:40  
    c=(4^(1-n)) * (z(n))^2; inner_sqrt=sqrt(1-c);  
    z(n+1)=(2^(n-0.5))*sqrt( 1-inner_sqrt );  
end  
rel_err_z=abs(z-pi)/pi;  
  
fprintf( '\n' );
```

Calcolo π : metodo 3

```
% METODO 3.  
y(1)=1;  
y(2)=2;  
for n=2:40  
    num=(2^(1/2)) * abs(y(n));  
    c=(4^(1-n)) * (z(n))^2;  
    inner_sqrt=sqrt(1-c);  
    den=sqrt(1+inner_sqrt);  
    y(n+1)=num/den;  
end  
rel_err_y=abs(y-pi)/pi;
```

```
% SEMILOGY PLOT.  
semilogy(1:length(u),rel_err_u,'k.',...  
          1:length(z),rel_err_z,'m+',...  
          1:length(y),rel_err_y,'ro');
```

Di seguito digitiamo sulla shell di Matlab/Octave

```
>> pigreco
```

Plot risultati

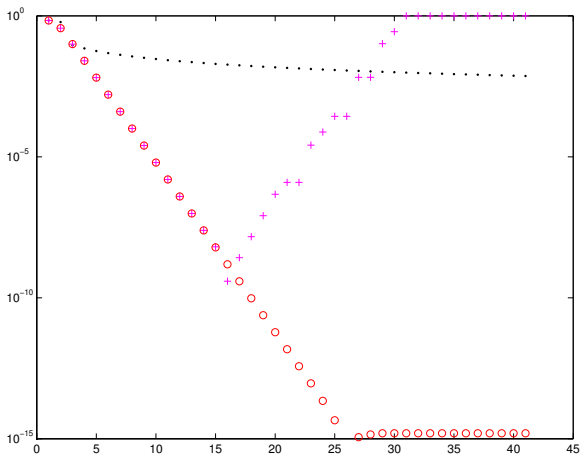


Figura : Grafico che illustra le 3 successioni, rappresentate rispettivamente da ., + e o.

- ▶ La prima successione converge molto lentamente a π , la seconda diverge mentre la terza converge velocemente a π .
- ▶ Per alcuni valori $\{z_n\}$ e $\{y_n\}$ coincidono per alcune iterazioni per poi rispettivamente divergere e convergere a π . Tutto ciò è naturale poichè le due sequenze sono analiticamente (ma non numericamente) equivalenti.
- ▶ Dal grafico dell'errore relativo, la terza successione, dopo aver raggiunto errori relativi prossimi alla precisione di macchina, si assesta ad un errore relativo di circa 10^{-15} (probabilmente per questioni di arrotondamento).

Una successione ricorrente.

Consideriamo la successione $\{I_n\}$ definita da

$$I_n = e^{-1} \int_0^1 x^n e^x dx \quad (4)$$

- ▶ $n = 0$: $I_0 = e^{-1} \int_0^1 e^x dx = e^{-1}(e^1 - 1)$.
- ▶ integrando per parti

$$\begin{aligned} I_{n+1} &= e^{-1} \left(x^{n+1} e^x \Big|_0^1 - (n+1) \int_0^1 x^n e^x dx \right) \\ &= 1 - (n+1) I_n. \end{aligned}$$

- ▶ $I_n > 0$, decrescente e si prova che $I_n \rightarrow 0$ come $1/n$.

Calcoliamo I_n per $n = 1, \dots, 99$:

- ▶ mediante la successione *in avanti*

$$\begin{cases} I_0 = e^{-1}(e^1 - 1) \\ I_{n+1} = 1 - (n+1) I_n. \end{cases} \quad (5)$$

- ▶ mediante la successione *all'indietro*

$$\begin{cases} t_{1000} = 0 \\ t_{n-1} = (1 - t_n)/n. \end{cases}$$

Si noti che se $I_{n+1} = 1 - (n+1) I_n$ allora $I_n = (1 - I_{n+1})/(n+1)$ e quindi $I_{n-1} = (1 - I_n)/n$.

Successioni ricorrente in Matlab

Scriviamo il codice in un file succricorrente.m.

```
% SUCCESSIONE RICORRENTE.  
clear all;  
% SUCCESSIONE " s_n " .  
s(1)=exp(-1);  
for n=1:99  
    s(n+1)=1-(n+1)*s(n);  
end  
% SUCCESSIONE " t_n " .  
M=1000;  
t=zeros(M,1); % INIZIALIZZAZIONE " t " .  
for n=M:-1:2  
    j=n-1;  
    t(j)=(1-t(n))/n;  
end
```

Successioni ricorrente in Matlab

```
% PLOT SEMI-LOGARITMICO.  
clf;  
semilogy(1:length(s),abs(s),'k-',...  
          1:length(s),abs(t(1:length(s)))));
```

Di seguito digitiamo sulla shell di Matlab/Octave

```
>> succricorrente
```

Plot risultati

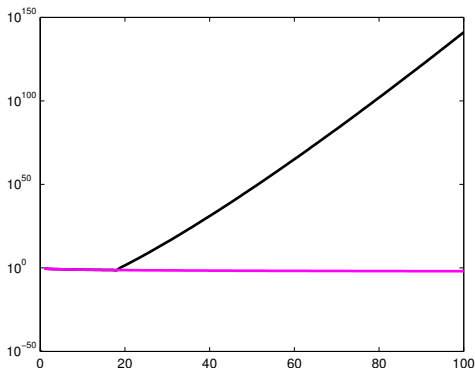


Figura : Grafico che illustra i valori assoluti assunti dalla successione in avanti (in nero) e all'indietro (in rosa magenta).

Commento successione ricorrente.

La successione in avanti amplifica gli errori. Infatti se

$$\bar{l}_0 = l_0 + \epsilon$$

$$\bar{l}_{n+1} = 1 - (n+1)\bar{l}_n$$

allora

$$\bar{l}_1 = 1 - \bar{l}_0 = 1 - (l_0 + \epsilon) = l_1 - \epsilon$$

$$\bar{l}_2 = 1 - 2\bar{l}_1 = 1 - 2(l_1 - \epsilon) = l_2 + 2 \cdot 1 \cdot \epsilon$$

$$\bar{l}_3 = 1 - 3\bar{l}_2 = 1 - 3(l_2 + 2\epsilon) = l_3 - 3 \cdot 2 \cdot 1 \cdot \epsilon$$

$$\bar{l}_4 = 1 - 4\bar{l}_3 = 1 - 4(l_3 + 3\epsilon) = l_4 - 4 \cdot 3 \cdot 2 \cdot 1 \cdot \epsilon$$

e in generale

$$\bar{l}_n = l_n + (-1)^n n! \cdot \epsilon.$$

con il termine $n! \cdot \epsilon$ che tende velocemente a $+\infty$ al crescere di n .

Commento successione ricorrente.

La successione all'indietro smorza gli errori. Infatti, se

$$\bar{l}_m = l_m + \epsilon$$

e

$$\bar{l}_{n-1} = (1 - \bar{l}_n)/n$$

allora

$$\bar{l}_{m-1} = (1 - \bar{l}_m)/m = (1 - (l_m + \epsilon))/m = l_m - \epsilon/m$$

$$\bar{l}_{m-2} = (1 - (l_{m-1} + \epsilon/m))/(m-1) = l_{m-2} - \epsilon/((m-1) \cdot m)$$

...

$$\bar{l}_{m-k} = \dots = l_{m-k} - \epsilon / \prod_{s=0}^k (m-s)$$

con il termine $\epsilon / \prod_{s=0}^k (m-s)$ che tende velocemente a 0 al crescere di k .

Facoltativo. Complessità : algoritmo di Horner.

Ci poniamo il problema di valutare il polinomio

$$p(x) = a_0 + a_1 \cdot x + \dots + a_n \cdot x^n \quad (6)$$

in un punto x .

Osserviamo che

$$p(x) = a_0 + x \cdot (a_1 + x \cdot (a_2 + \dots + x \cdot (a_{n-1} + x \cdot a_n))) \quad (7)$$

Supponiamo sia $a = (a_0, \dots, a_n)$ il vettore di dimensione $n + 1$ delle componenti del polinomio. Possiamo valutare il polinomio tramite i seguenti due algoritmi, il primo che valuta direttamente il polinomio secondo quanto descritto in (6), il secondo che effettua la stessa operazione come descritto in (7) calcolando dapprima $s_1 = a_{n-1} + x \cdot a_n$, poi $s_2 = a_{n-2} + x \cdot s_1$ e così via.

Complessità : algoritmo di Horner.

In Matlab avremo allora

```
function s=algoritmo1(a,x)
xk=1; s=a(1);
for i=2:length(a)
    xk=xk*x;
    s=s+a(i)*xk;
end
```

e

```
function s=algoritmo2(a,x)
L=length(a);
s=a(L); % COMPONENTE a_n IMMAGAZZINATA IN a(n+1).
for i=L-1:-1:1
    s=a(i)+x*s;
end
```

Complessità : algoritmo di Horner.

Se lanciamo il codice `demo_horner` per la valutazione di

$$p(x) = 1 + 2 \cdot x + 3 \cdot x^2 + 4 \cdot x^3 \text{ in } x = \pi$$

```
clear all;  
a=[1 2 3 4];  
x=pi;  
y1=algoritmo1(a,x);  
y2=algoritmo2(a,x);  
format long;  
y1  
y2
```

otteniamo

```
>> demo_horner  
ans = 1.609171052316469e+02  
y1 = 1.609171052316469e+02  
y2 = 1.609171052316469e+02  
>>
```


Complessità : algoritmo di Horner.

La differenza sta nella complessità computazionale e non nel risultato numerico. Il primo codice richiede $2n$ moltiplicazioni e n somme, mentre il secondo algoritmo n moltiplicazioni e n somme.