

Matlab: complessità e stabilità degli algoritmi. Alcuni esempi.

Ángeles Martínez Calomardo e Alvisé Sommariva

Università degli Studi di Padova

7 novembre 2012

Esempio di algoritmo instabile

Formula risolutiva dell'equazione di secondo grado

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

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

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

Valutiamo la 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);  
    s1=-p+u;  
else  
    fprintf('\n \t [RADICI COMPLESSE] ');  
    s1=[];  
end
```

Codice stabilità : algoritmo 2

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

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) & length(s1) > 0 & ...  
    length(t1) > 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]: 0.00000899999999772772  
[ALG.2] [1]: 0.00000900000000000000  
[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;
```

Calcolo π : plots

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

Di seguito digitiamo sulla shell di Matlab/Octave

```
>> pigreco
```

Plot risultati

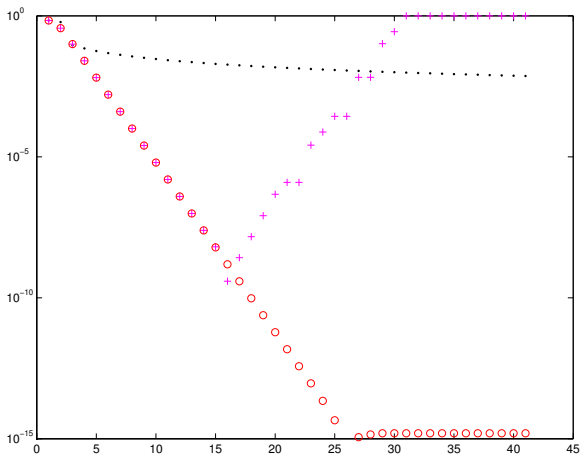


Figura : Errore relativo commesso con le 3 successioni, rappresentate rispettivamente da ., + e o.

- 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).

L'algoritmo 2 in dettaglio

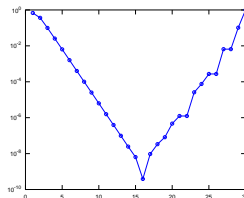
Successione approssimante π

Nell'approssimare il valore di π con la seguente formula ricorsiva

$$\begin{aligned}z_2 &= 2 \\z_{n+1} &= 2^{n-0.5} \sqrt{1 - \sqrt{1 - 4^{1-n} z_n^2}}, \quad n = 2, 3, \dots,\end{aligned}$$

si ottiene la seguente successione di valori (dove si è posto $c = 4^{1-n} z_n^2$).

$n+1$	c	$1 - \sqrt{1-c}$	z_{n+1}	$\frac{ z_{n+1} - \pi }{\pi}$
...
10	1.505e-04	7.529e-05	3.14157294036	6.27e-06
11	3.764e-05	1.882e-05	3.14158772527	1.57e-06
12	9.412e-06	4.706e-06	3.14159142150	3.92e-07
13	2.353e-06	1.176e-06	3.14159234561	9.80e-08
14	5.882e-07	2.941e-07	3.14159257654	2.45e-08
15	1.470e-07	7.353e-08	3.14159263346	6.41e-09
16	3.676e-08	1.838e-08	3.14159265480	3.88e-10
17	9.191e-09	4.595e-09	3.14159264532	2.63e-09
18	2.297e-09	1.148e-09	3.14159260737	1.47e-08
19	5.744e-10	2.872e-10	3.14159291093	8.19e-08
...
28	2.220e-15	1.110e-15	3.16227766016	6.58e-03
29	5.551e-16	3.330e-16	3.46410161513	1.03e-01
30	1.665e-16	1.110e-16	4.00000000000	2.73e-01
31	5.551e-17	0.000e+00	0.00000000000	1.00e+00
32	0.000e+00	0.000e+00	0.00000000000	1.00e+00



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 $I_{n-1} = (1 - I_n)/n$.

Successione 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
```

Successione ricorrente in Matlab

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

Di seguito digitiamo sulla shell di Matlab/Octave

```
>> succricorrente
```

Plot risultati

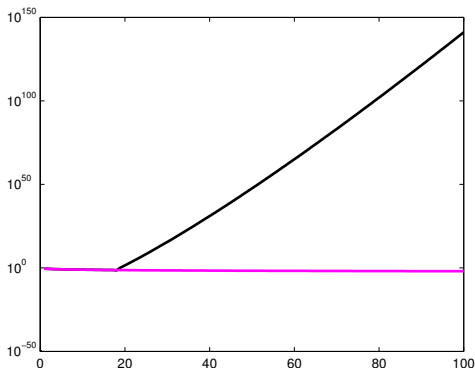


Figura : Grafico che illustra i valori assoluti assunti dalla successione in avanti (in nero) e all'indietro (in rosa magenta).

Instabilità della formula ricorsiva

La formula $I_n = 1 - n I_{n-1}$ è instabile, quindi amplifica l'errore ad ogni passo.

Infatti, nel calcolatore

$$(I_n + \varepsilon_n) = 1 - n(I_{n-1} + \varepsilon_{n-1}).$$

Sottraendo dalla precedente equazione la relazione $I_n = 1 - nI_{n-1}$ si può quantificare l'errore:

$$\varepsilon_n = -n \varepsilon_{n-1}, \quad \text{e per induzione} \quad |\varepsilon_n| = n! |\varepsilon_0|$$

Il fattore $n!$ amplifica l'errore di rappresentazione iniziale (su I_0), ε_0 .

- **Esempio.** Nel calcolo di I_{20} l'errore è $\varepsilon_{20} = 20! \varepsilon_0 \approx 2.7 \cdot 10^2$.

Formula alternativa stabile

La formula all'indietro smorza l'errore

Per l'errore al passo $n - 1$ si trova

$$\varepsilon_{n-1} = \frac{-1}{n} \varepsilon_n.$$

Partendo da m

$$|\varepsilon_{m-1}| = \frac{|\varepsilon_m|}{m}, \quad |\varepsilon_{m-2}| = \frac{|\varepsilon_m|}{m(m-1)}, \quad \dots, \quad |\varepsilon_{m-k}| = \frac{|\varepsilon_m|}{m(m-1) \cdots (m-k+1)}.$$

- La produttoria al denominatore abbatte rapidamente l'errore iniziale!
- Per esempio, per calcolare I_{25} partendo da $I_{40} = 0.5$, l'errore iniziale $|\varepsilon_{40}| < 0.5$ verrebbe abbattuto di un fattore

$$40 \cdot 39 \cdots 27 \cdot 26 = 5.2602 \cdot 10^{22}$$

Complessità : potenza di matrice.

Problema: calcolare la potenza p -esima di una matrice quadrata A di ordine n cioè

$$A^p := \underbrace{A * \dots * A}_{p \text{ volte}}$$

senza usare l'operatore di elevamento a potenza \wedge .

Primo algoritmo. Si può implementare il seguente pseudocodice

```
B=I ;  
for i=1:p  
    B=B*A ;  
end
```

in cui I è la matrice identica di ordine n e $*$ è il classico prodotto tra matrici.

Secondo algoritmo. Alternativamente (in maniera più stabile ed efficiente) si può decomporre p come

$$p = \sum_{i=0}^M c_i 2^i$$

ove $M = \lfloor \log_2 p \rfloor$ e $c_i = 0$ oppure $c_i = 1$. Si osserva facilmente che questa è la classica rappresentazione di p in base 2. Usando la proprietà della potenze

$$\begin{aligned} B &= A^p = A^{\sum_{i=0}^M c_i 2^i} = A^{\sum_{i=0}^M 2^i c_i} \\ &= (A^{2^0})^{c_0} * \dots * (A^{2^M})^{c_M} = \prod_{i=0}^M \left(A^{2^i} \right)^{c_i} \end{aligned} \quad (6)$$

ove ogni termine A^{2^i} può essere calcolato come $A^{2^{i-1}} A^{2^{i-1}}$.

Complessità : potenza di matrice.

Per capire il funzionamento dei metodi, introduciamo un ulteriore esempio esplicativo. Sia A una matrice di ordine 7 e $p = 22$.

Nel primo algoritmo si calcola A^p come

$$A^p = A * \dots * A$$

e quindi sono necessari 21 prodotti tra matrici.

Nel secondo algoritmo, essendo $p = 22$, Il valore assunto da c è quindi $[0\ 1\ 1\ 0\ 1]$ poiché, come si legge dalla rappresentazione binaria (da sinistra verso destra),

$$22 = 0 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot 2^3 + 1 \cdot 2^4.$$

Notiamo che il vettore c ha lunghezza

$M + 1 = \text{floor}(\log_2(22)) + 1 = 5$. Vogliamo quindi calcolare

$$B = (A)^0 * (A^2)^1 * (A^4)^1 * (A^8)^0 * (A^{16})^1. \quad (7)$$

Complessità : potenza di matrice.

Calcoliamo $B = (A)^0 * (A^2)^1 * (A^4)^1 * (A^8)^0 * (A^{16})^1$.

- Al primo passo, dobbiamo calcolare $B = (A)^0$. Poniamo $C = A$. Come suggerito, se inizialmente $B = I_7$, a questo punto non c'è nulla da fare poichè A ha ordine 7 e $A^0 = I_7$. A questa altezza abbiamo immagazzinato B , C ed A .
- Al secondo passo, ricordando (7), dobbiamo moltiplicare il valore di B ottenuto al passo precedente per $(A^2)^1$. Essendo $C = A$ osserviamo che $A^2 = A * A = C * C$ e quindi posto il nuovo C uguale a $C * C$ (per cui $C = A^2$) si ha $B = B * C$. A questa altezza abbiamo immagazzinato B , C ed A .
- Al terzo passo, ricordando (7), dobbiamo moltiplicare il valore di B ottenuto al passo precedente per $(A^4)^1$. Essendo $C = A^2$ osserviamo che $A^4 = A^2 * A^2 = C * C$ e quindi posto il nuovo C uguale a $C * C$ (per cui $C = A^4$) si ha $B = B * C$. A questa altezza abbiamo immagazzinato B , C ed A .

Complessità : potenza di matrice.

- Al quarto passo, ricordando (7), dobbiamo moltiplicare il valore di B ottenuto al passo precedente per $(A^8)^0$. Essendo $C = A^4$ osserviamo che $A^8 = A^4 * A^4 = C * C$. Poniamo il nuovo C uguale a $C * C$ (e quindi $C = A^8$). A questo punto non c'è nulla da fare poichè $(A^8)^0 = I_7$ e quindi $B = B * I_7$. A questa altezza abbiamo immagazzinato B , C ed A .
- Al passo $M = 5$, ricordando (7), dobbiamo moltiplicare il valore di B ottenuto al passo precedente per $(A^{16})^1$. Essendo $C = A^8$ osserviamo che $A^{16} = A^8 * A^8 = C * C$ e quindi posto il nuovo C uguale a $C * C$ (per cui $C = A^{16}$) si ha $B = B * C$. A questa altezza abbiamo immagazzinato B , C ed A .

Notiamo, che una volta finito il processo, A^{16} è immagazzinata in B .

Complessità : potenza di matrice.

Un codice che produce la decomposizione in potenze di 2 di un numero p è il seguente:

```
q=p;  
M=floor(log2(p))+1;  
c=[];  
for i=1:1:M  
    c(i)=mod(q,2);  
    q=floor(q/2);  
end
```

Complessità : potenza di matrice.

Uno pseudocodice che implementa il secondo algoritmo è

```
p=100; n=200;
c=trasforma_in_binario(p);
A=rand(n);
B=eye(n);
C=A;
M=floor(log2(p));
% B contiene la potenza di A finora calcolata.
% C contiene la potenza A^(2^index) finora calcolata.
for index=0:M
    j=index+1;
    if c(j) == 1
        B=B*C;
    end
    C=C*C;
end
```

Esercizio 1

Si implementino i due algoritmi proposti per il calcolo della potenza di matrice tramite due functions (senza usare l'operatore \wedge) e si calcoli l'errore relativo in norma infinito rispetto all'elevamento a potenza di MATLAB/OCTAVE per diverse matrici e potenze ($n = 25, 50$ e $p = 20, 40$).

Esercizio 2

Si confrontino poi i tempi di esecuzione delle due functions per il calcolo di A^{100} , con A matrice di numeri casuali di dimensione 200×200 .

Norme matriciali

- Nell'introduzione a MATLAB/OCTAVE abbiamo visto che il comando `norm(v)` calcola la norma del vettore v .
- Analogamente, per calcolare in MATLAB/OCTAVE la norma di una matrice A si usa il comando `norm(A)`.
- Se nessun ulteriore parametro viene specificato tale comando restituisce la norma 2 della matrice ovvero:

$$\|A\|_2 = \sqrt{\rho(A^T A)}$$

dove $\rho(A)$ è il raggio spettrale della matrice A . Altre possibilità sono:

- norma 1
 $\|A\|_1 = \max_j \sum_{i=1}^n |a_{ij}|$, in MATLAB/OCTAVE `norm(A,1)`;
- norma infinito
 $\|A\|_\infty = \max_i \sum_{j=1}^n |a_{ij}|$, in MATLAB/OCTAVE `norm(A,inf)`;
- norma di Frobenius
 $\|A\|_F = \sqrt{\sum_{i=1}^n \sum_{j=1}^n |a_{ij}|^2}$, in MATLAB/OCTAVE `norm(A,'fro')`;

Norme matriciali

Esempi

```
>>> A = [5 -4 2; 1 7 -6; 1 1 9]
```

```
A =
```

```
    5    -4     2  
    1     7    -6  
    1     1     9
```

```
>>> norm(A,1)
```

```
ans = 17
```

```
>>> norm(A,inf)
```

```
ans = 14
```

```
>>> norm(A, 'fro')
```

```
ans = 14.6287388383278
```

```
>>> norm(A)
```

```
ans = 12.0560586095913
```


Come misurare la durata di un programma

- Per confrontare due programmi che risolvono lo stesso problema è utile misurare il tempo di CPU (wallclock time) impiegato per eseguirli.
- In MATLAB/OCTAVE questo tempo si misura in secondi con il comando: `cputime`.

Esempio:

```
>> A = rand(5000);  
t = cputime;  
det(A);  
tfin=cputime;  
cpu=tfin-t  
  
>> cpu = 32.500
```

Facoltativo. Complessità : algoritmo di Horner.

Ci poniamo il problema di valutare il polinomio

$$p(x) = a_0 + a_1 \cdot x + \dots + a_n \cdot x^n \quad (8)$$

in un punto x .

Osserviamo che

$$p(x) = a_0 + x \cdot (a_1 + x \cdot (a_2 + \dots + x \cdot (a_{n-1} + x \cdot a_n))) \quad (9)$$

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 (8), il secondo che effettua la stessa operazione come descritto in (9) calcolando dapprima $s_1 = a_{n-1} + x \cdot a_n$, poi $s_2 = a_{n-2} + x \cdot s_1$ e così via.

Facoltativo: 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
```

Facoltativo: 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.