Laboratorio di Calcolo Numerico LEZ 2

Algebra vettoriale e matriciale, istruzioni condizionali e cicli
Soluzione di eq di 2° grado

Federico Piazzon

12 Aprile 2022

Outline

Matrici e vettori in Matlab

Vettori matrici e tensori

Morale: *In matlab tutto è una matrice*

Una variabile Matlab, indipendentemente dal tipo (o classe), ha *dimensione*: è *automaticamente* previsto che possa contenere molti dati *dello stesso tipo* organizzati come una tabella 0, 1, 2,... dimensioni.

- una variabile vuota ha dimensioni $[0, 0]$ (perchè la dimensionde di default è 2)
- un vettore riga ha dimensioni $[N, 1]$ colonna non riga
- un vettore riga ha dimensioni $[1, N]$
- una matrice ha dimensioni $[M, N]$
- possiamo definire tensori di dimensioni finite arbitrarie $[N_1, N_2, \ldots, N_n]$

definire vettori e matrici

- un variabile vuota:

```
>> w = []
```

- un vettore riga: componenti separate da **spazio** o da virgola

```
>> v = [ 1 2 3];
>> v = [ 1, 2, 3]
```

- un vettore colonna: componenti separate da **punto e virgola**

```
>> u = [4; 5; 6]
```

- un matrice va inserita riga per riga

```
>> A = [1 2 3; 4 5 6]
```

Dimensione di una variabile I

- il comando size che riporta le dimensioni di una variabile.
- il comando length che riporta la massima dimensione di una variabile. **Attenzione:** solo per i vettori corrisponde alla lunghezza.

```
>> v=[2,5,1]
v =
     2 5 1
>> length(v)
ans =
     3
>> size(v)
ans =
     1     3
```

Dimensione di una variabile II

```
>> A=[1 2 3;3 4 5];
>> length(A)
ans =
     3
>> size(A)
ans =
     2     3
```

comandi per creare vettori e matrici

- `x0:dx:x1` vettore riga $(x0, x0+dx, \ldots, x0+\lfloor (x1-x0)/dx \rfloor dx)$
- si può omettere `dx`, il default è 1
- `zeros(size1,size2,...)` crea variabile con componenti nulle e dimensioni `size1,size2,...`
- `ones(size1,size2,...)` crea variabile con componenti unitarie e dimensioni `size1,size2,...`
- `eye(N)` crea matrice identica [NxN]
- `diag(v)` con v vettore [1xN] o [Nx1] crea matrice [NxN] con v sulla diagonale
- `diag(v,k)` con v vettore [1xN-|k|] o [N-|k|x1] crea matrice [NxN] con v sulla *k*-esima sovra/sotto-diagonale

NB: `zeros(N)`, `ones(N)` sono equivalenti rispettivamente a zeros(N,N) e ones(N,N).

modifica vettori e matrici

- `'` trasposizione (N.B.:complessa, i.e., trasposta coniugata se valori complessi)
- `:` vettore colonna canonicamente (ordinamento crescente secondo l'ordine indotto dalla mappa $(i, j, h, k) \mapsto k + 10 \cdot h + 10^2 \cdot j + 10^3 \cdot i$) associato ad un tensore per una matrice A(3x2) da le colonne della matrice in ordine crescente in un'unica colonna
- `reshape(v,s1,s2,...` trasforma un vettore in tensore [s1 x s2 x...] se le dim sono compatibili
- `diag(A)` estrae la diagonale di A come vettore colonna
- `diag(A,k)` estrae la k-esima sopra/sotto-diagonale di A come vettore colonna
- `repmat(v,h,k,l,..)` replica la variabile v h volte in riga, k in colonna ecc ecc

Accesso alle componenti

Sia A una variabile di dimensioni $[N_1, N_2, \ldots, N_d]$.

singola componente

Per accedere alla componente in posizione `(i1,i2,...,id)` della variabile A (supp. $i_k \in \{1, 2, \ldots, N_k\}\ \forall k = 1, 2, \ldots, d$) si usa la sintassi

`A(i1,i2,...,id)` dalla riga i1, colonna i2, dimensione i3, ecc

riduzione del tensore

`A(v1,v2,...,vd)`

dove v1,...vd sono vettori di interi con componenti ciascuna in $\{1, 2, \ldots, N_d\}$,
Crea un tensore [`length(v1)`x...x`length(vd)`]

AAA: Matlab conta da 1 !

end nelle componenti

`end` assume il valore della dimensione k-esima della variabile A se è inserito nella posizione k-esima di A

```
>> A=[1 2 3;4 5 6;7 8 9];
>> A(2,2:end)      la riga 2 tutte le colonne dalla 2 alla fine

ans =

     5     6
```

keywords **end** e **:** II

: nelle componenti

: assume il valore del vettore degli indici della dimensione k-esima della variabile A se è inserito nella posizione k-esima.

```
>> A(1,:)

ans =

     1     2     3
```

definizione a blocchi I

E' possibile definire un vettore o matrice tramite concatenazione

```
>> u1=[1 2];
>> u2=[4 5 6];
>> [u1 u2 u1]
ans =
     1     2     4     5     6     1     2
```

attenzione alle dimensioni!

```
>> u1=[1 2];
>> u2=[4; 5; 6];
>> [u1 u2]
Error using horzcat
Dimensions of arrays being concatenated are not
    consistent.
```

definizione a blocchi II

NB: nelle matrici si inseriscono i blocchi per riga.

```
>> A=ones(2);
>> B=zeros(2);
>> C=[A;B,B;A]
Error using vertcat
Dimensions of arrays being concatenated are not
    consistent.
```

sbagliato ma inserisce a seguendo le righe in ordine poi
inserisce B e poi prova a mettere un'altra B affiancata

Operazioni con i vettori

Siano $u = (u_1, \ldots, u_n)$ e $v = (v_1, \ldots, v_n)$ vettori della **stessa dimensione** ed s uno scalare.

- `c=s*u`, prodotto dello scalare s con il vettore u,

$$c_1 = s \cdot u_1, \;\; c_2 = s \cdot u_2, \ldots, \;\; c_n = s \cdot u_n;$$

- `c=u'` trasposta del vettore u,
- `c=u+v` somma del vettore u col vettore v

$$c_1 = u_1 + v_1, \;\; c_2 = u_2 + v_2, \ldots, \;\; c_n = u_n + v_n;$$

- `c=u-v` differenza tra il vettore u e il vettore v

$$c_1 = u_1 - v_1, \;\; c_2 = u_2 - v_2, \ldots, \;\; c_n = u_n - v_n;$$

Op. componente a componente

scalari = array(1x1) vettori= matrici(1xn) o (nx1)

Attenzione!

Eseguibili solo se `size(u)=size(v)` non length

- `c=u.*v`, prodotto **componente a componente** del vettore u col vettore v

$$c_1 = u_1 \cdot v_1, \; c_2 = u_2 \cdot v_2, \ldots, \; c_n = u_n \cdot v_n;$$

- `c=u./v` divisione **componente a componente** del vettore u col vettore v,

$$c_1 = \frac{u_1}{v_1}, \; c_2 = \frac{u_2}{v_2}, \ldots, \; c_n = \frac{u_n}{v_n}.$$

- `c=u.^k` potenza k-sima **componente a componente** del vettore u con

$$c_1 = u_1^k, \; c_2 = u_2^k, \ldots, \; c_n = u_n^k.$$

Vari usi di * con vettori e matrici I

Attenzione alle dimensioni quando si usa *:

- prodotto matrice-vettore `A*v` con A e v matrice-vettore compatibili
- prodotto matriciale `A*B` con A e B matrici compatibili
- prodotto scalare `u*v’` con u v vettori riga compatibili

trasposizione ha precedenza su tutto a parte le parentesi

- prodotto tensorializzato `u*v’` con u e v vettori colonna compatibili

```
>> u=[1 2 3];v=[4 5 6];u1=u';v1=v';A=[u;v];B=[u1,v1];
u*v'
ans =

    32
```

B è una matrice 3x2 in quanto composta da 2 vettori colonna messi in "riga"

Vari usi di ∗ con vettori e matrici II

```
>> u1*v1'
ans =

     4     5     6
     8    10    12
    12    15    18
>> A*B
ans =

    14    32
    32    77
```

Istruzioni condizionali e cicli

Istruzione condizionale semplice

La struttura di un'istruzione condizionale semplice è la seguente:

```
    if <espressione logica>
         <processo>
    end
```

Ad esempio:

```
>> a=5;
>> s = 'Segno da determinare';
>> if a>0
    s = 'La variabile e' positiva';
end
>> s
s =
    'La variabile e' positiva'
```

Nel caso in cui a sia negativo il Matlab non assegnerà alcun valore alla variabile s. (Provatelo nella command window!)

Istruzione condizionale alternativa

Avremmo bisogno di una istruzione condizionale "alternativa" nel caso in cui volessimo assegnare un valore ad a se non rispetta la prima condizione. La struttura di un' istruzione condizionale alternativa è la seguente:

```
if <espressione logica>
    <processo 1>
else
    <processo 2>
end
```

Espressione logica

Vero	Falso
esegui proc. 1	esegui proc. 2

Istruzione condizionale multipla

Nella struttura condizionale alternativa si possono utilizzare nuovamente istruzioni condizionali: si parla in questo caso, di istruzione condizionale multipla.

```
    if <espressione logica 1>
        <processo 1>
    elseif <espressione logica 2>
            <processo 2>
    else
            <processo 3>
    end
```

$Exp.1 \backslash Exp.2$	Vero	Falso
Vero	esegui proc. 1	esegui proc. 1
Falso	esegui proc. 2	esegui proc. 3

Paragone elseif vs else if

I due programmi sono equivalenti

```
if <expr 1>
        <proc 1>
elseif <expr 2>
        <proc 2>
else
        <proc 3>
end
```

```
if <expr 1>
        <proc 1>
else
        if <expr 2>
                <proc 2>
        else
                <proc 3>
        end
end
```

NB: le istruzioni condizionali si possono innestare creando strutture anche molto complesse. Rispettare l'indentazione è importantissimo per rendere il codice leggibile.

Nel descrivere le istruzioni condizionali abbiamo bisogno delle espressioni logiche (`a<0`, `a>0`, ...)
Di seguito elenchiamo i simboli che permettono di eseguire espressioni logiche. Questi simbili si chiamano *operatori di relazione*:

==	uguale
~=	non uguale
<	minore
>	maggiore
<=	minore uguale
>=	maggiore uguale

Espressioni logiche complesse

Spesso dobbiamo combinale più espressioni logiche per definire un'istruzione condizionale (`a<0 e b< 50`, `a=0 o b=0,...`).
Per farlo si definiscono i seguenti *operatori logici*:

<table>
<tr><td>&&</td><td>and</td></tr>
<tr><td>||</td><td>or</td></tr>
<tr><td>∼</td><td>not</td></tr>
<tr><td>&</td><td>and (componente per componente)</td></tr>
<tr><td>|</td><td>or (componente per componente)</td></tr>
</table>

Un classico errore

Attenzione alla differenza tra $=$ e $==$! Consideriamo la variabile a=2 e il valore 1;

- $=$ assegna un valore ad una variabile:

```
>> a=2;
>> a=1 %equivale ad assegnare ad a il valore 1a =1
```

- $==$ ne controlla l'uguaglianza:

```
>> a==1 %equivale a chiedere a e' uguale a 1?
ans =
  logical
   0
```

Istruzioni logiche vettoriali

I test logici sono applicati per componenti e compatibili con le operazioni per componenti.

```
>> x=[0 2 -1.2 -2];
>> (x>-1.2).*(x<=0)
ans =
     1     0     0     0
>> max(x>0)
ans =
  logical
   1
>> min(x>=0)
ans =
  logical
   0
```

NB: E' più robusto `logical((x>-1.2).*(x<=0))`

Il commando switch

A volte risulta importante eseguire processi diversi al variare del valore di una variabile.
In Matlab è possibile usare il comando switch.

```
switch <espressione switch>
case <valore 1>
      <processo 1>
case <valore 2>
      <processo 2>
            . . . . . .
otherwise
      <processo altrimenti>
end
```

Il commando switch

Ad esempio, sappiamo che la funzione `sign(x)` assume valore 1 ($x > 0$), -1 ($x < 0$), 0 altrimenti. (`help sign`). Vogliamo scrivere una stringa esplicativa in ognuno dei casi:

```
>> a=1; s=sign(a);
>> % s = 1 se a > 0, s = -1 se a < 0, s = 0 se a = 0.
>> switch s
case 1
   stringa='a > 0';
case -1
   stringa='a < 0';
otherwise
   stringa='a = 0';
end
>> stringa
stringa =
a > 0
>>
```

Il ciclo for

Il ciclo *for* itera una porzione di codice al variare di certi indici

```
    for indice = range di variazione indice
        <processo>
    end
```

Ad esempio:

```
>> for i=1:10
i = i + 1
end
```

dove `i` è l'indice che varia da 1 a 10 .

L'indice `i` non deve necessariamente apparire all'interno del loop

```
>> s=1;
>> for i=1:5
    s = 2*s
end
```

Il ciclo while

Il ciclo *while* itera un processo finchè un'espressione logica è verificata

```
    while <espressione logica>
        <processo>
    end
```

Ad esempio:

```
>> b=5;
>> while b>0
    b = b - 1
end
```

Differenze tra *ciclo for* e *ciclo while*:

- Il *ciclo for* si usa quando si conosce il numero di volte che si vuole eseguire il ciclo mentre nel *ciclo while* si può non conoscere il numero di volte che si compie il ciclo;
- Il *ciclo for* NON ha bisogno di incrementare l'indice

Infatti:

```
>> c = 0
>> while c < 5
    c = c+1
end
```

```
>> d=0
>> for d=1:5
    d
end
```

Esercizi Proposti

Obbiettivo

Si vuole scrivere un algoritmo che risolva l'equazione di secondo grado a coefficienti reali

$$ax^2 + bx + c = 0 \qquad\qquad a, b, c \in \mathbb{R},$$

trovandone, se esistono, le due radici reali.

Obbiettivo

Si richiede di:

- Es 2.1 Considerare il caso in cui tutti i coefficienti sono NON nulli ($a \neq 0, b \neq 0, c \neq 0$) e implementare uno script che ne calcoli le radici in funzione del discriminante usando

  (2.1.a) Le formule instabili (script `eq2gr.m`);
  (2.1.b) Le formule stabili (script `eq2grst.m`);

  Scrivere poi due script `main2_1a.m` e `main2_1b.m` (si veda slides seguenti per il dettaglio) per testare gli algoritmi implementati.

- Es 2.2 Generalizzare lo script `eq2grst.m` al caso di coefficienti qualsiasi (considerare tutti i casi). Scrivere poi uno script `main2_2.m` (si veda slides seguenti per il dettaglio) per testare l'algoritmo implementato.

Esercizio 2.1.a

Ipotesi:

$a \neq 0, b \neq 0, c \neq 0.$

Sappiamo che le radici di un'equazione di secondo grado sono

$$x_1 = \frac{-b - \Delta}{2a} \quad x_2 = \frac{-b + \Delta}{2a}.$$

dove $\Delta = \sqrt{b^2 - 4ac}$ è il discriminante.

Scrivere uno script `eq2gr.m` seguendo le istruzioni della prossima slide

Traccia Script eq2gr.m

```
%%EQ2GR   Script per la risoluzione di un'equazione
%%        di secondo grado (solo soluzioni reali)
%%        con i coefficienti a, b e c non nulli
%%        FORMULE INSTABILI
%
%% Settare il formato di visualizzazione
%% Scrivere a video "Risoluzione eq. secondo grado"
%% Chiedere all'utente di inserire i coefficienti a, b, c
%% Controllare che siano tutti NON nulli
%% se è così calcolare le radici,
%% altrimenti dare un messaggio di errore
%%CALCOLO DELLE RADICI
%% calcolo del discriminante delta
%% se delta<0 nessuna sol. reale (output video)
%% se delta = 0 due sol. reali coincidenti(output video)
%% altrimenti x1 e x2 reali e distinte (output video)
```

Vogliamo testare `eq2gr.m` sui seguenti dati.

a	b	c	x_1	x_2
1	10^{-5}	-2×10^{-10}	-2×10^{-5}	10^{-5}
-10^{-7}	$1+10^{-14}$	-10^{-7}	10^{7}	10^{-7}
10^{-10}	-1	10^{-10}	10^{10}	10^{-10}

Script chiamante

A tal fine prepariamo uno script chiamante `main2_1a.m` che:

- richieda (con input()) il valore di `a,b,c,x1vera,x2vera` del caso da considerare
- esegua un controllo su $a \neq 0$ $b \neq 0$ $c \neq 0$ ed in caso negativo esca con messaggio di errore (si veda `help error`)
- esegua `eq2gr.m` ricavando `x1` e `x2`
- calcoli gli errori relativi (output video)

Esercizio 1.b - Fromule stabili

- Creare un secondo script `eq2grstab.m` ottenuto modificando `eq2gr.m` in modo da implementare le formule stabili. (Modificare l'intestazione (header) e i commenti coerentemente)

NB: unica modifica nel caso $\Delta > 0$:

Formule stabilizzate

$$x_1 = -\frac{b + sign(b)\sqrt{\Delta}}{2a} \qquad x_2 = \frac{c}{a x_1}.$$

- Creare uno script chiamante `main2_1_b.m` per effettuare il test di `eq2grstab.m` sui medesimi dati.

Esercizio 2.2 - Coefficienti qualsiasi **pseudocodice**

Si crei un altro script `eq2grstab_all.m`

```
Se a == 0
    Se b == 0
        x1=NaN; x2=NaN;
        Output video con fprintf
    altrimenti
        x1 = -c/b; (Equazione di grado 1)
        x2=x1;
        Output video con fprintf
    fine
altrimenti
    Calcolo i discriminante
    se il discriminante < 0
        x1=NaN; x2=NaN; Output video con fprintf
    altrimenti se il discriminante == 0
         x1=x2 = -b/(2a), Output video con fprintf
    altrimenti
        se b == 0
         x1=... x2=...., Output video con fprintf
        altrimenti
          Formule stabili
          Output video con fprintf
        fine
    fine
fine
```

Test

Si testi sui seguenti dati con un opportuno script chiamante `main2_2.m` ottenuto dalla modifica di `main2_1a.m`.

a	b	c	x_1	x_2
1	2	3	/	/
3	8	2	-2.3874	-0.27924
2	4	2	-1	-1
0	1	2	-2	/
3	5	0	-1.6667	0
4	0	3	/	/
4	0	-3	0.86603	-0.86603
0	0	2	/	/
3	0	0	0	0
0	0	0	/	/
1	0	-4	-2	2

Esercizio 2.3 facoltativo

main2_3.m

Modificare `main2_2.m` per ottenere uno script test chiamante con stampa dei risultati su file. NB: il settaggio di fprintf deve essere tale da ottenere un output ordinato tipo tabella (i.e., i numeri devono essere incolonnati).

NB: per la stampa su file consultare `doc fprintf` e la slide seguente

Stampa dei risultati su file

E' possibile stampare su file tramite la funzione fprintf con la sintassi

```
>> f_id=fopen('nomefile.txt','permesso');
>> fprintf(f_id,'stringa');
>> fclose(f_id)
```

dove la stringa permesso ha i valori

'r'	Open file for reading.
'w'	Open or create new file for writing. Discard existing contents, if any.
'a'	Open or create new file for writing. Append data to the end of the file.
'r+'	Open file for reading and writing.
'w+'	Open or create new file for reading and writing. Discard existing contents, if any.
'a+'	Open or create new file for reading and writing. Append data to the end of the file.
'A'	Open file for appending without automatic flushing of the current output buffer.
'W'	Open file for writing without automatic flushing of the current output buffer.