UNIVERSITÀ
DEGLI STUDI
DI PADOVA

DIPARTIMENTO
DI INGEGNERIA
INDUSTRIALE

DIPARTIMENTO
MATEMATICA
DIPARTIMENTO DI MATEMATICA "TULLIO LEVI-CIVITA"

MATLAB STEP BY STEP

Materiale realizzato da Michela Redivo Zaglia

con il contributo di E. Bachini, L. Bruni, W. Erb, A. Larese, F. Piazzon

UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA INDUSTRIALE

DIPARTIMENTO MATEMATICA

DIPARTIMENTO DI MATEMATICA "TULLIO LEVI-CIVITA"

Laboratorio di Calcolo Numerico LAB 3

Strutture algoritmiche, Function m-file

Docenti: E. Bachini, L. Bruni

Email: elena.bachini@unipd.it Email: bruni@math.unipd.it

20 marzo 2024

Outline

Istruzione condizionale semplice

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

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

Ad esempio (da command window):

```
>> 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

Nel caso in cui volessimo assegnare un valore ad `s` se la condizione risulta falsa, dovremo utilizzare una istruzione condizionale alternativa la cui struttura è

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

Esempio:

```
>> a=5;
>> s = 'Segno da determinare';
>> if a>0
     s = 'La variabile e'' positiva';
   else
     s = 'La variabile e'' negativa o nulla';
   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 seguenti due codici sono equivalenti (ma nel secondo caso la `if <expr 2>` non fa parte di una struttura condizionale multipla e quindi necessita di una `end`.

```
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.

Espressioni logiche semplici

Per descrivere le espressioni logiche (`a<0`, `a>0`, ...) che daranno per risultato la costante logica 1 (= Vero) oppure 0 (= Falso), abbiamo bisogno di poter confrontare i valori (abitualmente numerici) descritti da espressioni.
Di seguito elenchiamo i simboli che permettono di costruire le espressioni logiche. Questi simboli si chiamano *operatori di relazione*:

$==$	uguale
$\sim=$	non uguale (diverso)
$<$	minore
$>$	maggiore
$<=$	minore uguale
$>=$	maggiore uguale

Espressioni logiche complesse

Spesso dobbiamo combinare tra loro più espressioni logiche per definire la nostra condizione (`a<0` `e` `b< 50`, `a=0` `o` `b=0`,...).

Per farlo si definiscono i seguenti *operatori logici*:

&&	and (per valori scalari logici)
||	or (per valori scalari logici)
$\sim$	not
&	and (per array, agisce componente per componente)
|	or (per array, agisce componente per componente)

Un classico errore

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

= assegna un valore ad una variabile:

```
>> a = 1 % equivale ad assegnare ad a il valore 1"
a =
     1
```

== ne controlla l'uguaglianza dei valori:

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

Istruzioni logiche per array (vettori o matrici)

Quando si lavora con array (vettori o matrici) i test logici sono applicati componente per componente.

```
>> x=[0 2 -1.2 -2];
>> (x>-1.2).*(x<=0)  % espressione logica che restituisce un
>>                   % vettore numerico (0 oppure 1)
ans =
     1     0     0     0
>> max(x>0)          % restituisce un valore logico
ans =
  logical
   1
>> min(x>=0)         % restituisce un valore logico
ans =
  logical
   0
```

ATTENZIONE: Quando si opera con array è meglio utilizzare la funzione `logical` con argomento l'espressione che si desidera valutare logicamente in quanto restituisce un vettore logico.

```
>> x=[0 2 -1.2 -2];
>> logical((x>-1.2).*(x<=0))
ans =
  1x4 logical array
   1   0   0   0
```

Il commando `switch`

A volte risulta importante eseguire processi diversi a seconda del valore assunto da una certa variabile (spesso un valore numerico intero o una stringa di caratteri). In questi casi per evitare di dover utilizzare troppe strutture `if` è possibile usare la struttura `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 (creiamo uno script di nome `segno.m`)

```
a = input ('Dammi il valore di a ');
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
disp (stringa);
```

Lo eseguiamo:

```
>> segno
>> Dammi il valore di a -2
a < 0
>>
```

Il ciclo for

Il ciclo *for* (detto anche ciclo fisso) ripete un certo numero di istruzioni, per un numero prefissato di volte. Il numero di volte è determinato dal rango di variazione di un indice.

```
    for indice = range di variazione indice
        <processo dipendente dal valore dell'indice>
    end
```

Ad esempio:

```
for i = 1:5
   j = i + 2  % j varia in funzione del valore della variabile
              % indice i.
              % j assume successivamente i valori 3, 4, 5, 6, 7
end
```

`i` è l'indice del ciclo, che varia da 1 a 5 con incremento (o passo) 1. L'indice `i` non deve necessariamente apparire all'interno del loop (anche se normalmente ciò accade).

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

Il ciclo while

Il ciclo *while* (o ciclo con condizione) ripete un certo numero di istruzioni, fintantoché una certa espressione logica (condizione) risulta verificata, ovvero è vera.

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

Ad esempio:

```
b = 5;
while b > 0
    b = b - 1 % ad ogni iterazione il contenuto della variabile b
              % diminuisce di 1
end
```

ATTENZIONE: Il blocco all'interno del ciclo deve obbligatoriamente, ad un certo punto, rendere falsa la condizione. In caso contrario si innesca il cosiddetto loop infinito e si deve interrompere l'esecuzione (CTRL-C).

Esempio:

```
b = 5; c = b;
while b > 0
    c = c - 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 (il numero di volte è fissato e predeterminabile) mentre nel *ciclo while* non si conosce a priori il numero di volte in cui il ciclo verrà eseguito, perché dipende dalla condizione.

- Nel *ciclo for* NON è necessario incrementare l'indice (sarebbe un errore): l'incremento viene realizzato in modo automatico. Nel *ciclo while* se si vuole tenere il conto di quante volte viene eseguito, bisogna usare una variabile (detta contatore) che va inizializzata al di fuori del ciclio ed incrementata al suo interno.

Infatti:

```
ind = 0
while c^2-a/b < 0
    ......
    ind = ind + 1;
    ......
end
```

```
for ind = 1:2:10
    ......
end
```

function m-file - Caratteristiche principali

Un **m-file di tipo function** è un programma matlab che implementa un algoritmo, che, alla *chiamata* della function,

- viene eseguito utilizzando solamente i *parametri di ingresso fittizi* della function e restituisce i soli *parametri di uscita fittizi*;
- prevede (opzionalmente) la creazione e l'utilizzo di *variabili locali* (i.e., interne alla function stessa) che non influenzano quelle *globali* (i.e., presenti nel workspace),
- I valori da assegnare ai parametri di ingresso della function sono determinati dai valori *parametri di ingresso attuali* indicati nella chiamata (i nomi possono anche essere diversi). I valori da resituire ai *parametri di uscita attuali* sono determinati dai valori che all'interno della function sono stati attribuiti ai *parametri di uscita*.
  L'associazione viene gestita in modo automatico in base alla corrispondenza posizionale. tra parametri attuali e fittizi.

Come definire una function m-file

```
function [out_1,out_2,...,out_m]=nomefunction(in_1,in_2,...,in_n)
%UNTITLED Summary of this function goes here
%   Detailed explanation goes here
out_1=...
out_2=...
...
out_m=...
end % (opzionale)
```

- Come gli script, la function m-file deve essere salvata in un file il cui nome deve coincidere con quello indicato nell'intestazione (`nomefunction.m`);
- nella function posso usare solo variabili presenti nei parametri di ingresso indicati nell'intestazione;
- posso creare variabili locali senza influenzare/sovrascrivere quelle presenti nel workspace;
- i nomi dei vari output devono essere diversi tra loro;
- i nomi dei vari input devono essere diversi tra loro;
- tutti i parametri di uscita devono essere assegnati.

Struttura del file function

1. il file deve iniziare con l'istruzione di intestazione contenente obbligatoriamente la keyword `function` e con l'indicazione dei parametri di input e di output:
   `function [listaout] = nomefunction (listainp)`
2. Help (sotto forma di commento)
   - descrizione della function, della sua chiamata e (opzionale) dell'algoritmo implementato
   - descrizione dei parametri di input fittizi (se vettori/matrici indicare le dimensioni)
   - descrizione dei parametri di output fittizi (se vettori/matrici indicare le dimensioni)
3. corpo della function: implementazione dell'algoritmo (usare i commenti per descrivere quanto si sta facendo)

NB: `help nomefunction` visualizza tutto il primo blocco di commenti del file `nomefunction.m` presente nella cartella di lavoro.

Come "chiamare" una function - Sintassi

In uno script (o in un'altra function m-file) la chiamata (ovvero la richiesta di esecuzione) avviene inserendo il comando

```
[myout_1,...,myout_k] = nomefunction(myin_1,...,myin_n);
```

- tutti[1] i parametri di input attuali devono essere definiti;
- solo i contenuti degli output a cui assegno un nome vengono modificati a seguito dell'esecuzione della function (i.e., può essere $k < m$);
- se l'output manca completamente, viene creata la variabile `ans` contenente il **primo** output della function;
- posso usare nomi per input e output **diversi** da quelli usati in `nomefunction.m`, conta solo la loro **posizione** nella lista.

[1] Di norma, vedremo in seguito le eccezioni.

Esempio variabili globali/locali 1

Definita la function `myexp.m`

```
function y = myexp(x)
e = exp(1);
y = e.^x;
end
```

Ne effettuiamo la chiamata da workspace (oppure da uno script)

```
>> t=1.2;
>> z = myexp(t)
z =
    3.3201
>> e
Unrecognized function or variable 'e'.
```

NB: Le variabili `y` ed `e` sono state create e/o assegnate a livello locale nella function e non sono nel Workspace! Quindi non sono conosciute.

Esempio variabili globali/locali 2

Supponiamo che nella Current Folder ci sia il file `myfun.m` Se nel Workspace è presente la variabile globale `x` e viene chiamata la function

```
function z = myfun(t)
z = x.^t;
end
```

tramite

```
s = myfun(2)
```

Matlab restituirà un errore perchè la *variabile locale* `x` in `myfun` non è stata assegnata e non compare nei parametri di ingresso indicati nell'intestazione (la *variabile globale* `x` non è "vista" da `myfun`!)

```
Unrecognized function or variable 'x'.
Error in myfun (line 2)
z=x.^t
```

Utilizzo tipico delle function

- **Implementazione algoritmo** (m-file di tipo function) che implementa un algoritmo sui dati di input per calcolare l'output. Nelle function m-file non devono esserci istruzioni `input` o di visualizzazione di risultati. Possono solo esserci istruzioni `disp` di segnalazione eventuale.

- **Programma Chiamante** (m-file di tipo script) definisce tutte le variabili dell'esperimento (leggendole da tastiera (o file) ed assegnandole a delle variabili globali), chiama la/le funzione/i volute per avere restituiti gli output. Salva o visualizza i risultati e eventualmente fornisce un output grafico/video.

ATTENZIONE: Il file che contiene la function m-file e lo script che la vuole eseguire, devono essere nella stessa cartella!

Esempio

```
function y = polinomiosecondogrado(x,a,b,c)
%-------------------------------------------------------------------
% polinomiosecondogrado valuta il polinomio p(x)= a x^2+b x+c
%-------------------------------------------------------------------
% INPUT
% x     vettore [1 X N] o [N X 1], ascisse di valutazione
% a     scalare, coeff 2o grado
% b     scalare, coeff 1o grado
% c     scalare, termine noto.
% OUTPUT
% y     vettore con size(y)=size(x), valori del polinomio p
%-------------------------------------------------------------------
y = (a*x+b).*x+c;
```

Si può chiamare con lo script

```
a = 2; b = 1; c = -3;
xmin = -2; xmax = 4;
x = linspace(xmin,xmax);
y = polinomiosecondogrado(x,a,b,c);
plot(x,y)
```

Passare funzioni come input: function handle

In matlab c'è una distinzione tra

- la funzione da applicare ai parametri di input: **la function**
- la funzione come *oggetto* astratto: **il function handle**, una speciale *classe* in matlab.

Uso dei function handle:

- `@nomefunction` crea il function handle della function `nomefunction.m`
- un function handle può essere passato come input ad un altra funzione
- la conversione function handle $\rightarrow$ function è invece automatizzata da matlab quando si valuta un input un handle.

NB: quando creiamo la anonymous function `f=@(x) x.^2`, la variabile `f` è in realtà un function handle, questo spiega anche il nome.

Esempio function handle

Creiamo il file nestedfun.m e myfun1.m

```
function ff = nestedfun(f,x)
ff = f(f(x));
end
```

```
function y = myfun1(x)
y = exp(x+1);
end
```

Nella command window definiamo esplicitamente una anonymous function

```
myfun2 = @(x) exp(x+1);
```

Otteniamo lo stesso risultato con i seguenti due gruppi di comandi

```
x = 0:0.1:1
y = nestedfun(@myfun1,x)
```

```
x = 0:0.1:1
y = nestedfun(myfun2,x)
```

viceversa otterremo un errore. Si noti che per passare come argomento una function m-file, dobbiamo mettere il carattere @, mentre quando la function è una anonymous function definita nello stesso file, non serve.

[Facoltativo:] input ed output opzionali

Matlab supporta la definizione di funzioni che hanno parametri di ingresso e/o di uscita opzionali:

```
function [out,varargout]=nomefunction(in1,in2,varargin)
```

In questa sintassi `varargin` e `varargout` sono variabili di tipo cell

- varargin e varargout possono in realtà essere usati per definire più input ed output opzionali
- varargout si usa quando è possibile nell'argomento della function calcolare gli altri output senza aver calcolato quelli opzionali.
- per il controllo di quanti in/output sono stati utilizzati dal programma chiamante si unsano `nargout` e `nargin`.

Si vedano: `help varargin`, `help varargout`, `help nargin`, `help nargout`.