INTERFACCIA UTENTE E STRINGHE

Per visualizzare su schermo il valore di una variabile **a** senza farla precedere dalla scritta **a=** , si usa il comando **disp** (abbreviazione di *display*). Per esempio

```
» a=pi/2;
» disp(a)
    1.5708
```

Il comando **disp** funziona per variabili di qualunque classe e di qualunque formato, ma è in grado di scrivere solo una variabile alla volta.
Per visualizzare con un solo comando due variabili già definite **a** e **b** , occorre quindi inserirle in un'unica variabile. Per esempio se **a** e **b** fossero scalari, si potrebbe scrivere

```
» disp([a b])
» disp([a;b])
```

per visualizzarle in riga o in colonna.

Benché **MatLab** lavori principalmente con numeri, esiste la possibilità di trattare stringhe alfanumeriche, con le quali è anche possibile gestire una buona interfaccia con l'utente.
Quindi oltre alle variabili di classe **double** (numeri in doppia precisione) e alle variabili di classe **logical** (generate dagli operatori relazionali), esistono le variabili di classe **char**, cioè stringhe di caratteri. Per esempio

```
» s='ciao'
```

definisce una variabile **s** di classe **char**.
Le stringhe vanno sempre racchiuse tra due apici singoli (e non tra virgolette doppie).

Dato che non è possibile combinare assieme variabili di classe **char** con variabili di altra classe, per scrivere su una stessa riga una stringa di caratteri e un numero occorre qualche accorgimento.
Per esempio per scrivere la frase "La variabile a vale 6.2832" (dove $a = 2\pi$) occorre convertire **a** in stringa mediante la funzione **num2str** (che si legge "number to string") che trasforma un numero in una stringa di caratteri.
Quindi si può formare una matrice riga **s** costituita da stringhe (quindi di classe **char**), che affianchi la frase e il numero **a** convertito in stringa.

```
» a=2*pi;
» s=['La variabile a vale ',num2str(a)];
» disp(s)
La variabile a vale 6.2832
```

La funzione **num2str** di norma visualizza al massimo 4 cifre dopo il punto decimale, ma ha anche un argomento opzionale per il massimo numero di decimali. Usando opportunamente questa funzione è possibile ottenere una buona interfaccia utente.
Illustriamo la procedura con questo esempio

```
» a=[pi/2 , 2.3];
» s=['La matrice a e'' formata da ', num2str(a(1,1),5) , ...
' e ',num2str(a(1,2))])
» disp(s)
La matrice a e' formata da 1.57080 e 2.3
```

Commento:
- Viene definita una matrice 1×2 **a** .
- Si definisce la matrice 1×4 **s** i cui elementi sono quattro stringhe.
- Le quattro stringhe di **s** vanno racchiuse tra parentesi quadre **[ ]** e separate da virgole (o spazi).
- La prima stringa è "La matrice a e' formata da ".

  La stringa contiene un apice. Per inserirlo nella stringa, senza che venga considerato come il delimitatore della stringa, l'apice va raddoppiato.
- La seconda stringa è il numero $\pi/2$ (cioè $a(1,1)$) trasformato in stringa con 5 cifre decimali con **num2str** .

  Dato che la definizione di **s** non sta per intero nella riga, è possibile continuarla nella riga seguente andando a capo mediante la serie di tre punti **...**
- La terza stringa è " e ", cioè la congiunzione e racchiusa tra due spazi.
- La quarta stringa è il numero 2.3 (cioè $a(1,2)$), trasformato in stringa mediante **num2str** .
- Viene dato il comando **disp** il cui argomento è racchiuso tra parentesi tonde.

La funzione **num2str** risolve in modo relativamente elementare alcuni problemi di visualizzazione, ma la soluzione migliore è quella di usare la funzione **sprintf** che è mutuata direttamente dal linguaggio C.
Le opzioni della funzione sono tantissime e in molti casi piuttosto complicate, per cui rimandiamo ai manuali di C per la descrizione di tutte le possibilità, limitandoci a un esempio con le opzioni più semplici.

La sintassi è

```
» sprintf('xxxx', elenco variabili)
```

dove **xxxx** è una stringa, detta stringa di formato. Questa stringa fornisce, con apposito codice, le istruzioni per visualizzare le variabili che seguono. Dopo la virgola va dato un elenco di una o più variabili, separate da virgole.
Esempio

```
» x=2; y=[3.56890988 , 3]; z='questi sono numeri reali'
» s=sprintf('Si ha x = %d e y= %f %f\n%s',x,y,z);
» disp(s)
Si ha x = 2 e y= 3.568910 3.000000
questi sono numeri reali
```

La stringa di formato di **sprintf** comprende
• due stringhe di testo: "Si ha x= " , " e y= "
• la stringa di formato **%d** che indica un numero intero e si applicherà alla prima variabile in elenco, cioè **x** .
• le due stringhe di formato **%f** che indicano un numero decimale e si applicheranno ai due elementi della seconda variabile, cioè **y** .
• la stringa speciale **\n** che indica ritorno a capo.
• la stringa di formato **%s** che indica una stringa e si applicherà alla terza variabile, cioè **z** .
Osservazioni:
• **sprintf** è una funzione che restituisce una stringa, in questo caso **s** , che può essere visualizzata mediante il comando **disp** .
• In mancanza di altre opzioni, un numero decimale viene scritto con sei cifre decimali.
• Il numero 2 viene visualizzato senza parte decimale come conseguenza dell'uso di **%d** , mentre il numero 3 viene scritto con sei cifre decimali perché si è usato **%f** .

• Se una delle variabili è una matrice, la formattazione viene applicata a tutti i suoi elementi, letti per colonne, senza interruzioni.

Oltre a quelli visti, i principali codici sono:
%e : formato in notazione scientifica
la stringa speciale **\t** che indica il carattere di tabulazione

Inoltre è possibile specificare il numero di decimali. Per esempio **%0.3f** specifica notazione decimale con 3 cifre dopo il punto decimale.

INTERFACCIA CON L'ESTERNO

È possibile salvare l'intero workspace (cioè tutte le variabili definite fino a quel momento) mediante il comando

```
» save mievariabili.mat
```

dove "mievariabili" è un nome di fantasia. Il suffisso consigliato è ".mat". Le variabili salvate nel file "mievariabili" potranno essere recuperate successivamente con il comando

```
» load mievariabili.mat
```

che recupera tutte le variabili salvate. Se qualche variabile è stata ridefinita, il valore letto nel file viene sostituito a quello corrente. È possibile anche salvare solo parte del workspace nel seguente modo

```
» save mievariabili.mat x y z ...
```

dove **x y z** etc. è l'elenco delle variabili da salvare separate da spazi (e non da virgole).
Nel file "mievariabili.mat" le variabili sono salvate con codifica binaria, per cui il file non è leggibile in modo elementare da un altro programma applicativo.
Per salvare variabili in un file leggibile da un programma esterno (per esempio un editore di testo o un foglio elettronico tipo Excel), si deve aggiungere l'opzione **-ascii** . Il comando

```
» save alcunevariabili.txt x y z -ascii
```

salva i valori delle variabili **x,y,z** nel file "alcunevariabili.txt", leggibile da qualunque editore di testo.
Osserviamo che, usando questa opzione, si salvano solo i valori delle variabili, ma se ne perdono i nomi.
Il comando **save** ha altre opzioni. Per vederle tutte usare **help save**

È inoltre possibile recuperare una serie di dati da un file di testo creato da un programma esterno.
Come esempio, creiamo con qualunque editore di testo (anche con Word, pur di salvare il file come puro testo con suffisso .txt) un file così fatto

1	1.98	56
2.6	−.17	0

in cui gli elementi di ogni riga sono separati mediante spazi o mediante il carattere TAB e le righe sono separate con il carattere RETURN. Salviamo quindi i dati in un file di puro testo con un nome di fantasia, per esempio "matr.txt"

in una folder conosciuta da **MatLab** .
Eseguendo il comando

```
» load matr.txt
```

viene creata nel workspace la variabile `matr` come matrice 2×3 con i valori forniti.
Per concludere accenniamo alla funzione `xlsread` che consente di recuperare i dati da uno spreadsheet creato con Excel. La funzione

```
» a=xlsread('dati.xls')
```

crea una nuova variabile `a` che contiene i dati del file "dati.xls". Ovviamente occorre che il foglio elettronico sia strutturato in un modo da avere dati gestibili da **MatLab** .

GLI M-FILE

Dovendo eseguire più volte una serie di comandi complessi, non occorre ricopiarli ogni volta. Si può invece scrivere la lista dei comandi in uno "script" che va registrato in un apposito file detto M-file. Lo script può poi essere richiamato quando necessario.

Per esempio se volessimo moltiplicare tutti gli elementi della diagonale di una matrice quadrata `a` per `5` potremmo eseguire i seguenti comandi

```
» d=diag(a)
```

che crea la matrice colonna contenente gli elementi della diagonale di `a` . Poi

```
» d1=diag(d)
```

che crea una matrice diagonale che ha la diagonale identica a quella di `a` . Poi

```
» e=a-d1
```

che crea una matrice `e` con diagonale nulla e per il resto identica ad `a` . Poi

```
» d2=d1*5
```

che moltiplica la diagonale di `d1` per `5` , ottenendo `d2` . Infine

```
» a1=d2+e
```

costruisce la matrice voluta. Il risultato finale è la matrice `a1` .

Se vogliamo eseguire molte volte questa procedura, possiamo creare un M-File (c'è il comando apposito nel Menu "File") e inserire la lista delle istruzioni in questo modo

```
1 |   d=diag(a);
2 |   d1=diag(d);
3 |   e=a-d1;
4 |   d2=d1*5;
5 |   a1=d2+e
```

(i numeri compaiono automaticamente e servono per il "debugging").
Conviene porre dei segni `;` alla fine di ogni riga del file tranne l'ultima, affinché eseguendo il comando non compaiano sul display i risultati intermedi delle matrici `d, d1, d2, e` , ma compaia solo `a1` . Indi si salva il file per esempio come " `diagocinque.m` " (suffisso `.m` obbligatorio) in una directory (folder) di quelle che il programma **MatLab** riconosce come valide nel suo elenco di "Path" o nella sua folder di default che appare nella parte alta dello schermo.
Ogni volta che si scriverà

```
» diagocinque
```

MatLab eseguirà la serie di istruzioni dell'M-file **sulla matrice** `a` proprio come se li avessimo scritti ogni volta. Le matrici `d, d1, d2, e, a1` rimangono allocate nel *workspace* corrente, ovvero vengono definite o vengono ridefinite, se già esistevano, e possono essere richiamate.

DEFINIZIONE DI NUOVE FUNZIONI IN MATLAB

Naturalmente si vorrebbe poter eseguire la serie di comandi su una qualsiasi matrice senza che questa abbia necessariamente nome `a` . Per questo occorre definire un nuova funzione **MatLab** .
Questo si può fare mediante un M-File di tipo diverso che definisce una nuova funzione ed è così congegnato:

```
1 |   function y = diagoper5(a)
2 |   % diagoper(a) moltiplica la diagonale di a per cinque
3 |   d=diag(a);
4 |   d1=diag(d);
5 |   e=a-d1;
6 |   d2=d1*5;
7 |   y=d2+e;
```

Si noti il "cappello" `function y = diagoper5(a)` .

L'istruzione **function** serve a fornire tutti i dati relativi alla nuova funzione definita:
Questa istruzione, che deve essere la prima del file, svolge i seguenti compiti:

• dichiara che il file contiene una funzione e non uno script e che quindi le variabili usate rimagono locali.
• dichiara il nome della nuova funzione definita, in questo caso **diagoper5** . La funzione viene aggiunta all'elenco di funzioni che **MatLab** conosce.

L'M-File va salvato con il nome " **diagoper5.m** " **che deve essere identico** al nome dato alla funzione nella prima riga del listato.

• dichiara il numero e il nome delle variabili di input. In questo caso è una sola e ha nome **a** .
• dichiara il numero e il nome delle variabili di output. In questo caso è una sola e ha nome **y** .
Ogni volta che si calcola la funzione

```
» diagoper5(x)
```

su una qualunque matrice quadrata **x** , si ottiene come **ans** la matrice ottenuta quintuplicando la diagonale di **x** .

Altre osservazioni:
• Le righe del listato precedute dal simbolo **%** sono commenti ad uso di chi scrive e non vengono eseguite. Ma i commenti opzionali posti subito dopo la parola **function** sono ad uso dell'utilizzatore della funzione. Infatti appaiono nella finestra di comando quando si richiede aiuto per la funzione col comando **help**

```
» help diagoper5

  diagoper(a) moltiplica la diagonale di a per cinque
```

• La differenza fondamentale tra un M-file di tipo "script" e uno **di tipo funzione** è il fatto che in quest'ultimo le variabili sono "locali". Per esempio, nella funzione ora definita, le variabili usate **a, d, d1, d2, e, y** (comprese quindi quelle di input e output) vengono cancellate appena terminata l'esecuzione della funzione e non interferiscono con le variabili definite dall'utente nel *workspace* che possono anche avere gli stessi nomi.

• Si noti che al risultato finale occorre dare il nome **y** perché **y** è il nome scelto per la variabile di output. Se **y** non viene definita all'interno dell'M-file, il risultato della funzione va perduto.

• Dato che la variabile **y** è locale, conviene porre il simbolo **;** anche alla fine dell'ultima istruzione per evitare che il risultato compaia due volte sul display, una volta come variabile locale, la seconda come risultato della funzione.

• Alla funzione va dato un nome di fantasia, in questo caso si è scelto **diagoper5** .

• Occorre badare che il nome scelto non sia già il nome di un comando **MatLab** o di un M-File già esistente.
Nel primo caso l'esistente comando **MatLab** non sarebbe più riconosciuto, a favore della nuova funzione definita dall'utente.
Nel secondo caso l' M-File già esistente andrebbe perso, se salvato nella stessa folder, oppure potrebbe non essere riconosciuto a favore del nuovo comando, se salvato in altra folder.

Le funzioni possono avere anche molti argomenti. Per esempio, la funzione precedente può essere generalizzata, in modo che abbia due argomenti in entrata:
- la matrice su cui operare
- il numero per cui moltiplicare la diagonale che può essere qualunque e non solo 5.
Per ottenere questo occorre modificare le seguenti istruzioni

```
1 |   function y = diagoper(a,num)
2 |   % diagoper(a) moltiplica la diagonale di a per num
    ----------------------------
6 |   d1=d1*num;
```

Le funzioni possono inoltre avere diverse variabili in uscita. Per esempio, la funzione precedente può essere modificata, in modo che fornisca, oltre alla matrice con la diagonale cambiata, anche la matrice diagonale con la diagonale originale (che, nell'esecuzione, è la variabile temporanea **d1**).
Andrebbe solo modificata la prima riga del file.

```
1 |   function [y,d1] = diagoper(a,num)
```

Eseguendo però la funzione si otterrebbe solo la matrice modificata, cioè solo la prima variabile di output. Per ottenerle entrambe, al momento di calcolare la funzione vanno specificate entrambe le variabili a cui assegnare i due risultati, per esempio così

```
» [m,n]=diagoper(b,t)
```

La matrice modificata sarà **m** , mentre **n** sarà la matrice con la diagonale originale.

IL CICLO for...end

Esistono molti comandi per creare M-File assai complessi e versatili e strutture logiche simili a quelle usate nei co-

muni linguaggi di programmazione, tipo **`for...end , while...wend , if...then...else`** con ampia gamma di opzioni.
Uno dei cicli più usati è il loop **`for...end`** .
Dovendo per esempio eseguire il comando $R_j \to R_j - (a(j,1))/a(1,1))R_1$ (primo passo dell'algoritmo gaussiano) sulle righe di una matrice **`a`** di 10 righe si dovrebbero eseguire le nove istruzioni

```
» a(2,:)=a(2,:)-(a(2,1)/a(1,1))*a(1,:);
» a(3,:)=a(3,:)-(a(3,1)/a(1,1))*a(1,:);
.......
» a(10,:)=a(10,:)-(a(10,1)/a(1,1))*a(1,:);
```

Conviene quindi creare un M-File con loop del tipo **`for`** :

```
1 |   for index=2:10
2 |        a(index,:)=a(index,:)-(a(index,1)/a(1,1))*a(1,:);
3 |   end
```

Il comando **`for`** fa eseguire tutte le istruzioni successive fino al comando **`end`** facendo assumere a **`index`** successivamente i valori $2, 3, ..., 10$.
L'editore degli M-Files provvede in generale ad indentare automaticamente tutti i comandi che appaiono tra **`for`** e **`end`** e che fanno parte del loop, cioè a spostarli a destra in modo che il file sia più leggibile e il loop rimanga evidenziato. Questo accade però solo se le istruzioni vengono inserite nell'ordine. Quando un M-file viene modificato, spesso le indentazioni automatiche non vengono create. È buona norma tenere comunque indentati i loop.
Se si vuole che **`index`** assuma i valori $2, 4, 6, ..., 20$ la sintassi sarà

```
1 |   for index=2:2:20
```

Se si vuole che **`index`** assuma i valori 1 , 1.1 , 1.2, ..., 3 , la sintassi sarà

```
1 |   for index=1:0.1:3
```

Se si vuole che **`index`** assuma successivamente i valori $10, 9, 8, ..., 0$ la sintassi sarà

```
1 |   for index=10:-1:0
```

I loop **`for...end`** possono essere "nidificati", cioè inseriti uno dentro l'altro. Per esempio, se si vuole una funzione che crei una matrice quadrata **`n x n`** in cui l'elemento di posto **`(i,j)`** sia **`i*j+2`** , si può creare una funzione del tipo seguente che dipende solo da **`n`** .

```
1 |   function a = pippo(n)
2 |
3 |   for jndex=1:n
4 |        for index=1:n
5 |             a(index,jndex)=index*jndex+2;
6 |        end
7 |   end
```

Notare i due loop dipendenti da due variabili diverse **`index`** e **`jndex`** , il loop in **`index`** nidificato dentro al loop in **`jndex`** e la doppia indentazione che rende più visibili i due loop.

Come nome per le variabili di loop abbiamo usato **`index`** e **`jndex`** , invece che **`i`** e **`j`** , per non confonderle con l'unità immaginaria.

IL COMANDO if...end

Il test condizionale con **`if...end`** viene usato quando si vuole che alcune istruzioni vengano eseguito solo se è verificata una certa condizione. Il test con **`if`** viene usato quasi esclusivamente negli M-files, anche se sarebbe possibile usarlo nella riga di comando. La sintassi è

```
if [test]
   [istruzioni da eseguire se il risultato non è 0]
else
   [istruzioni da eseguire se il risultato è 0]
end
```

Di solito il test che segue il comando **`if`** è un operatore relazionale che restituisce il valore **`1`** se il test è vero e il valore **`0`** se è falso. Quindi la prima serie di istruzioni è eseguita solo se la relazione è vera.
Le istruzioni che seguono **`else`** sono eseguite se la relazione è falsa. Il comando **`else`** è opzionale e può essere omesso. In tal caso, se la relazione è falsa, non viene fatto niente.
Esempio 1: Se si vuole eseguire direttamente l'algoritmo di Gauss su una matrice, si potrebbe voler evitare di eseguire l'operazione $R_3 \to R_3 - cR_1$ su una matrice **`a`** quando il numero **`c`** è zero.
Il comando per l'operazione elementare è:

```
» a(3,:)=a(3,:)-c*a(1,:)
```

Quindi si possono dare le seguenti istruzioni:

```
if c~=0
    a(3,:)=a(3,:)-c*a(1,:)
end
```

Esempio 2: Si vogliono cambiare in 0 tutti gli elementi non reali di una matrice riga `x`. È possibile creare mediante un M-File una funzione che esegua questa operazione per ogni matrice riga. La funzione seguente agisce su una matrice riga e ne crea un'altra seguendo appunto questa regola.

```
1 |   function y=zeroc(x)
2 |
3 |   s=size(x,2);
4 |   for index=1:s
5 |        if imag(x(index))~=0
6 |             y(index)=0;
7 |        else
8 |             y(index)=x(index);
9 |        end
10|   end
```

Osservazioni sulle righe del listato:

1 La funzione ha un nome di fantasia `zeroc`. Il nome della variabile di input è `x`, quello della variabile di output è `y`. Notiamo che, dato che l'M-file è una funzione, `x` e `y` sono variabili locali, cioè non interferiscono con le variabili definite nel *workspace*.

3 La funzione `size(x,2)` calcola il numero di colonne della matrice `x` per sapere quante volte va eseguito il loop.

4-10 Questo è il loop `for...end` che esamina tutti gli elementi della matrice riga `x`. La variabile di controllo del loop è stata chiamata `index`.

5-9 Questo è il comando `if...else...end`. L'istruzione 6 viene eseguita se la parte immaginaria dell'elemento di `x` non è zero (cioè se l'elemento non è reale). In caso contrario viene eseguita l'istruzione 8 che ricopia l'elemento di `x` nel corrispondente di `y`.

Esiste un'altra opzione per la struttura `if...else...end` ed è l'uso di `elseif` al posto di `else` o prima di `else`.
Benché possibile, l'uso di `elseif` è sconsigliato a favore del comando `switch` che viene descritto in seguito.

ERRORI E AVVERTIMENTI

Quando **MatLab** incontra un errore nell'eseguire un comando o una funzione, interrompe l'esecuzione e scrive un messaggio di errore.
Per esempio, se si tenta di calcolare l'inversa di una matrice non quadrata `x`, si ottiene

```
» inv(x)
???  Error using ==> inv
Matrix must be square.
```

Ogni altro calcolo viene interrotto. Se l'errore avviene all'interno di un M-file, la sua esecuzione viene interrotta.
In casi meno gravi, se l'esecuzione non pregiudica il proseguimento del calcolo, **MatLab** lancia solo un avvertimento (warning) e non interrompe i calcoli.
Per esempio se si tenta di calcolare l'inversa di una matrice quadrata, ma non invertibile `x`, si ottiene

```
» inv(x)
Warning:  Matrix is singular to working precision.
```

I calcoli proseguono, ma **MatLab** avverte che probabilmente i risultati sono inaffidabili.

È possibile simulare errori e avvertimenti all'interno di un M-File coi comandi seguenti

```
error('questa operazione non e'' lecita')
warning('questa operazione puo'' dare risultati errati)
```

Il comando `error` scrive il messaggio di errore e interrompe l'esecuzione dell'M-file.
Il comando `warning` scrive il messaggio di avvertimento, ma non interrompe i calcoli. In pratica è come un comando `disp` con qualche opzione in più (vedi `help warning`).
Per esempio nell'M-file `diagoper` creato sopra come esempio, sarebbe bene evitare che l'utenti usi la funzione su una matrice non quadrata.
Questo si può ottenere inserendo questa istruzione subito dopo il "cappello" `function` + stringa di aiuto

```
[r,c]=size(a);
if r~=c
     error('matrice non quadrata')
end
```

Se si tenta di usare la funzione su una matrice non quadrata, l'esecuzione si arresta e compare la scritta "matrice non quadrata".

GLI OPERATORI LOGICI

Spesso gli operatori relazionali `==`, `<=` etc. sono usati congiuntamente agli operatori logici booleani AND e OR che in **MatLab** sono generati mediante i simboli seguenti: `&` per AND `|` per OR
Esempi:

```
if a>0 & a<=5
   [istruzioni]
end
```

le istruzioni verranno eseguite solo se $0 < a \leq 5$

```
if a<=0 | a>5
   [istruzioni]
end
```

le istruzioni verranno eseguite solo se a è esterno all'intervallo $(0, 5]$

```
if (a>0 & a<1)| a>2
   [istruzioni]
end
```

le istruzioni verranno eseguite solo se a è nell'intervallo $(0, 1)$ o nell'intervallo $(2, +\infty)$.

OPERATORI RELAZIONALI E MATRICI

Occorre qualche cautela nell'usare il comando `if...end` con gli operatori relazionali, quando ci si riferisce a matrici e non a scalari, dato che gli operatori relazionali confrontano tutti gli elementi delle matrici. Per esempio se si vogliono eseguire certe istruzioni solo sulle matrici simmetriche, la sintassi corretta è

```
if a==a.'
   [istruzioni da eseguire se a è simmetrica]
end
```

Se invece si vogliono eseguire certe istruzioni solo su matrici non simmetriche, la sintassi seguente non funziona

```
if a~=a.'
   [istruzioni da eseguire se a non è simmetrica]
end
```

Le istruzioni non verranno mai eseguite, perché l'operatore relazionale `a~=a.'` restituisce sempre una matrice non completamente fatta di 1. Conviene quindi una delle seguenti due sintassi:

```
if a==a.'
else
   [istruzioni da eseguire se a non è simmetrica]
end
```

Oppure, sfruttando la funzione `any`:

```
if any(a~=a.')
   [istruzioni da eseguire se a non è simmetrica]
end
```

IL COMANDO switch...case...end

Benché il test `if...end` sia a prima vista il test condizionale più naturale, è molto più conveniente in molti casi (anzi è consigliato da molti programmatori e da molte norme) il ciclo `switch...end`
Come esempio creiamo un M-file che interpreta un numero intero `a` da 0 a 10 come voto e fornisce il giudizio.

```
if a==10
     disp('ottimo')
elseif a==8 | a==9
     disp('buono')
elseif a<6
     disp('scarso')
else
     disp('sufficiente')
end
```

Confrontare la leggibilità del listato sopra col seguente che fa le stesse cose

```
switch a
case 10
     disp('ottimo')
case {8,9}
     disp('buono')
case {0,1,2,3,4,5}
     disp('scarso')
otherwise
     disp('sufficiente')
end
```

Purtroppo il ciclo **switch...end** non prevede la sintassi **case <6** che è invece usata in altri linguaggi di programmazione, costringendo, in certi casi, a strutture più complesse.
Esiste comunque un trucco, non riportato in generale dai manuali, per aggirare l'ostacolo ed è quello di usare per **switch** una variabile che valga 1

```
vero=1;
switch vero
case a==10
     disp('ottimo')
case (a==8 | a==9)
     disp('buono')
case a<6
     disp('scarso')
otherwise
     disp('sufficiente')
end
```

Osserviamo comunque che nel ciclo **switch...end** vengono eseguite solo le istruzioni che seguono **il primo** test **case** che soddisfa il criterio richiesto e non vengono considerati eventuali **case** successivi che soddisfano il criterio, diversamente dal linguaggio C. Per esempio

```
switch a
case 3
   [istruzione 1]
case {3,4}
   [istruzione 2]
end
```

Se **a** vale 3, verrà eseguita l'istruzione 1 e non l'istruzione 2. Se **a** non è né 3 né 4, non verrà eseguito niente.

IL CICLO while...end

Questo ciclo viene usato quando non è possibile prevedere a priori la lunghezza di un loop, ma si vuole che questo venga eseguito fintantoché è verificata (o non è verificata) una certa condizione.
La sintassi è la seguente

```
while [condizione]
   [istruzioni da eseguire fintantoché la condizione è valida]
end
```

Consideriamo a titolo di esempio la successione ricorsiva così definita: $a_1 = 1 \quad ; \quad a_{n+1} = \sqrt{a_n^2/(a_n+2)}$
Notoriamente il limite della successione è 0. Si vuole determinare il primo n per cui $a_n < 10^{-5}$.
Si può usare uno script così fatto

```
n=1;
a=1;
while a>=1e-05
     a=sqrt(a^2/(a+2));
     n=n+1;
end
disp(n)
```

- Vengono inizializzati l'indice **n=1** e il primo elemento della successione **a=1** .
- L'iterazione con **while** viene eseguita fintantoché **a** è maggiore o uguale a 10^{-5}.
  Notare il numero 10^{-5} inserito usando la notazione esponenziale.
- Ad ogni passo viene calcolato il nuovo **a** e viene incrementato l'indice **n** .
- Quando **a** è minore di 10^{-5} il ciclo **while** non viene più eseguito e viene visualizzato l'indice **n** in corrispondenza del quale è terminato il ciclo.

È possibile usare lo stesso script con un'altra successione di cui non si conosce il limite. Se la successione non converge a 0, il ciclo **while...end** può durare indefinitamente.
Conviene allora fissare un tetto massimo per le iterazioni, per esempio $n = 1000$, e stabilire di uscire comunque dal ciclo **while...end** quando **n** arriva a **1000** .
Questo si realizza col comando **break** . Il ciclo va così modificato.

```
while a>1e-05
     a= ------- % altra successione
     n=n+1;
     if n==1000
          break
     end
end
```

Il comando **break** consente di uscire dal ciclo **while...end** .
Il comando **break** può essere usato anche all'interno di un ciclo **for...end** o di uno **switch...end** e, in caso di loop nidificati, fa uscire solo da quello al cui interno è stato inserito.