Premessa

IDL è un linguaggio orientato all'analisi e alla visualizzazione dei dati. I dati possono rappresentare segnali audio, immagini a livelli di grigio, immagini a colori, elaborazioni statistiche e altro. La potenza di IDL deriva dal fatto che il linguaggio permette di manipolare i dati ad alto livello, in particolare gli array, come se fossero oggetti semplici, senza la necessità di entrare nei dettagli, evitando l'uso di istruzioni di iterazione per accedere ai singoli elementi. Inoltre è dotato di funzioni di manipolazione dei dati (per esempio per l'image processing) e di visualizzazione degli stessi che permettono di scrivere programmi di poche righe di codice, estremamente potenti; in questo modo è possibile concentrarsi sul cosa fare piuttosto che sul come implementarlo.

Esempio: la seguente funzione di 1 riga di codice dà l'idea della potenza del linguaggio (la funzione calcola il modulo del gradiente di una immagine a livelli di grigio).

```
function gradient, image       ; calcola |d/dx| + |d/dy|
        return, abs ( image − shift ( image, 1, 0 ) ) + abs ( image − shift ( image, 0, 1 ) )
end
```

IDL è un linguaggio interpretato, quindi un programma sorgente scritto in IDL viene eseguito, un'istruzione alla volta, da un interprete. L'interprete IDL è inserito in un ambiente di sviluppo denominato *IDLDE* (IDL Development Environment) che è dotato di funzioni di editing.

Ambiente di sviluppo di IDL: IDLDE

L'ambiente di programmazione di IDL è un'applicazione MDI (multiple document interface), costituito da una serie di menù di comandi con sottomenù a tendina, delle toolbars, un editor di programmi (in grado di gestire diversi documenti contemporaneamente), una finestra di output (output log), una riga per l'immissione di comandi, denominata prompt dei comandi di IDL, e dalla barra di stato.

L'editor dell'ambiente di sviluppo permette di scrivere programmi da far eseguire all'interprete. Però è anche possibile immettere comandi, attraverso il prompt di IDL, direttamente interpretati ed eseguiti dall'interprete.

La finestra di output non è la finestra in cui sono riportati tutti gli output; piuttosto sono ripetuti i comandi immessi al prompt dei comandi, riportati gli errori che si verificano a run−time, e stampati gli output relativi alle funzioni *print* e *help*, due funzioni molto utili per indagare il tipo e il contenuto delle variabili. In particolare, come vedremo, per visualizzare dati e immagini si utilizzano delle istruzioni, molto potenti, che aprono delle finestre sul desktop e poi disegnano al loro interno.

Può essere utile dare un'occhiata al menù *File/Preferences*. La voce *Graphics* permette di selezionare il tipo di refresh video: conviene impostare il valore di RETAIN, nella sezione *Backing store*, uguale a 2; in questo modo IDL spende un po' di tempo per copiare ogni immagine stampata a video in opportuni buffer di memoria, e così facendo IDL si prende cura di ridisegnarle, se necessario. Un'altra voce utile è la voce *Editor*: la sezione C*ompiling* permette di decidere se IDL deve, una volta chiesta la compilazione di un progetto che è stato modificato dall'ultima sua apertura, avvisare che il progetto è cambiato e chiedere se salvarlo su disco (si possono risparmiare un paio di secondi di tempo per richiesta evitata da moltiplicare per il numero di compilazioni che si devono effettuare ✳).

L'ambiente di sviluppo di IDL non è un ambiente visuale, però è possibile programmare applicazioni con interfacce grafiche, anche molto avanzate, con l'unico svantaggio che non c'è alcun supporto per farlo; in altre parole è necessario scrivere tutto il codice in dettaglio, per creare le finestre, i bottoni, per gestire gli eventi e le procedure di risposta agli eventi, ecc. (Questo è in contrasto con la capacità di IDL di manipolare i dati ad un alto livello di astrazione: qui si richiede un basso livello di dettaglio!)

IDLDE, permette di editare veri e propri programmi, e pure di immettere comandi al prompt di IDL. In genere i comandi immessi al prompt non hanno una grande strutturazione, p.e. è difficile giustificare l'immissione di un'istruzione di selezione quale può essere una *if−then−else*, e inoltre sono finalizzati ad un output a breve termine. Inoltre, il prompt dei comandi tiene traccia di tutti i comandi immessi dall'utente (recuperabili premendo il tasto *freccia−su*); quindi è possibile far riferimento ad una variabile definita in precedenza.

Prime esperienze al prompt di IDL

L'uso del prompt dei comandi non è da sottovalutare; è molto utile soprattutto in fase di progettazione di un algoritmo, anche se questo è piuttosto complesso e si sviluppa in più procedure/funzioni; infatti, se ad un certo punto si blocca un algoritmo (volontariamente, mediante il comando *stop*, o per un errore di run−time) si può procedere "a vista", cioè immettendo comandi al prompt dei comandi.

Le funzioni PRINT e HELP

Le funzioni *print* e *help* sono tra le prime funzioni del linguaggio utili da imparare.

Print

La funzione *print* permette di stampare sulla finestra di output una costante (numerica o alfanumerica) o il contenuto di una variabile. Vedremo molti esempi d'uso di questa funzione.

Help

La funzione *help* permette di ottenere informazioni a riguardo della sessione di lavoro corrente o indagare sul tipo di una variabile. Vedremo molti esempi d'uso anche di questa funzione.
Nota: Questa funzione è anche utile per monitorare lo stato e la quantità di memoria heap utilizzata; in questo caso la sintassi da usare è la seguente: `IDL > help, /memory`

Nomi validi per le variabili

Le variabili possono avere nomi lunghi da un minimo di 1 carattere ad un massimo di 255 caratteri. Il primo carattere deve essere una lettera o il carattere _ (underscore). Le variabili possono contenere lettere, cifre numeriche e il carattere $ (dollar). IDL non distingue tra maiuscolo e minuscolo. Inoltre una variabile non può chiamarsi come una funzione, una procedura o una parola riservata di IDL.

Esempi di nomi di variabili validi: ***myVar*** , ***var1*** , ***read6_$*** , ***only8bit*** , ***_32bit***
Esempi di nomi di variabili non: **1st_v*ar*** , ***my−var*** , ***32bit*** , ***last.name*** , ***a&b***

Dati di tipo semplice

Di seguito sono riportati i tipi di dato semplice previsti da IDL e, di fianco, le funzioni che permettono di forzare una variabile ad un preciso tipo di dato (operatori di cast).

Undefined	–
Byte (8 bit senza segno)	Byte
Integer (16 bit con segno)	Fix
Long integer (32 bit con segno)	Long
Floating point a singola precisione (32 bit)	Float
Floating point a doppia precisione (64 bit)	Double
Complex a singola precisione	Complex
Complex a doppia precisione	DComplex
String (da 0 a 32767 caratteri)	String

Nota: Un complex (o un dcomplex) è, in sostanza, una coppia di valori floating point (a singola o doppia precisione). Per estrarre la parte reale di un numero complesso si usa la funzione (operatore di cast) *FLOAT*, mentre per estrarre la parte immaginaria si usa la funzione *IMAGINARY*. Inoltre la funzione *ABS* calcola il modulo del numero complesso.

Attenzione: Il tipo di una variabile si decide al momento della sua inizializzazione, in altre parole non è necessario dichiarare il tipo di una variabile prima di definirla.

Esempi:

```
IDL > var1= 10B              | var1 è una variabile di tipo byte
IDL > var2= 10               | var2 è una variabile di tipo integer
IDL > var3= 10L              | var3 è una variabile di tipo long integer
IDL > var4= 10.0             | var4 è una variabile di tipo floating point a singola precisione
IDL > var5= 10.0D            | var5 è una variabile di tipo floating point a doppia precisione
IDL > var6= complex(1.0,2.5) | var6 è una variabile di tipo complex (il primo valore indica la
                             | parte reale, il secondo valore indica la parte immaginaria)
IDL > var7= complex(1.0)     | var7 è una variabile di tipo complex (la parte reale vale 1.0
                             | mentre la parte immaginaria vale 0.0)
IDL > va8= dcomplex(1.0,2.5) | va8 è una variabile di tipo complex a doppia precisione
IDL > var9= 'first string'   | var9 è una variabile di tipo string
```

Nota: Non è necessario dichiarare la dimensione di una stringa. Inoltre, le stringhe possono essere comprese tra apici sempilici (') oppure doppi apice (").

```
IDL > var10= "second string" | var10 è una variabile di tipo string
IDL > var11= "Don't write!"  | var11 è una variabile di tipo string
IDL > var12= 'Don''t write!' | var12 è una variabile di tipo string
IDL > var13= 32000           | var13 è una variabile di tipo integer
IDL > var14= 33000           | var14 è una variabile di tipo long integer
```

Attenzione: Definita una variabile, questa assume un preciso tipo di dato, però è sufficiente una nuova assegnazione per cambiare il tipo di dato; l'interprete non segnala nessun tipo di errore.

Esempi:

```
IDL > varA = 0               | varA è una variabile di tipo integer
IDL > varA = 10.0            | varA è ora una variabile di tipo floating point
IDL > varA = 'Hello world'   | varA è ora una variabile di tipo string
```

Esercizio 1: Utilizzare l'istruzione *help*, dopo ognuna delle precedente istruzione, per verificare che la variabile *varA* cambia di tipo.

Vediamo ora qualche esempio semplice di come si possono usare gli operatori di *cast*:

```
IDL > var = FLOAT(1)          | var è un float che ha valore 1.0
IDL > var = COMPLEX(1)        | var è un complex che ha valore (1.0,  0.0)
IDL > var = 10.0 + FLOAT(2)   | var è un floating point che ha valore 12.0
```

Conversioni automatiche

Quando un'espressione contiene variabili di tipo differente, IDL converte ***automaticamente*** le variabili al fine di preservare la massima precisione nei risultati. Questo come regola generale.

Esempio: `IDL > x = 1.1 + 2` | l'*int* 2 è convertito in *float* prima dell'operazione

Molte di queste conversioni automatiche possono indurre qualche perplessità. Ne vedremo dei casi esemplari. In generale è bene esplicitare le conversioni, mediante gli operatori di cast:

Esempio: `IDL > x = 1.1 + float(2)`

Vediamo ora degli esempi di conversione, alcune dei quali… "esemplari":

```
IDL > a= 3 / 2 * 5.0          | a è un floating point che vale 5.0
IDL > a= ( 3 / 2 ) * 5.0      | a è un floating point che vale 5.0
IDL > a= ( 3 / 2. ) * 5       | a è un floating point che vale 7.5

IDL > a= 33000                | a è un long integer che vale 33000
IDL > b= fix(a)               | b è un integer che vale –32536,
                              | nessun warning è stampato !!!

IDL > a= -7                   | a è un integer
IDL > b= byte(a)              | b è un byte che vale 251
                              | nessun warning è stampato !!!

IDL > a= complex(1.5, 2.7)
IDL > b= float(a)             | b è un floating point che vale 1.5

IDL > str= '100 Euro'         | str è la stringa '100 Euro'
IDL > b= str * 2.             | b è un floating point che vale 200.0

IDL > str= '123.33'
IDL > b= str + 3              | b è un intero che vale 126

IDL > str= 'Euro 20.5'
IDL > b= str * 2.             | viene segnalato un errore (e meno male!)
```

Esercizio 2: Verificare, sulla finestra di log, il risultato dell'istruzione seguente:

```
IDL > help, 2 + ( 7 / 2. ) * ( complex ( 2, 2 )  / 2 )
```

Dati di altro tipo

Oltre ai dati di tipo semplice IDL supporta i seguenti tipi di dato:

Strutture	Una struttura è un'aggregazione di diversi tipi di dato
Vettori e Matrici	Un array è una collezione di elementi, tutti dello stesso tipo
Puntatori	Un puntatore è un riferimento ad una variabile costruita in memoria heap

Strutture

Le strutture sono generalmente utilizzate per raggruppare informazioni anche di tipo diverso. In IDL ci sono due tipi di strutture: con nome e senza nome (anonime). In entrambi i casi le strutture si definiscono mediante l'uso delle parentesi graffe, e le strutture con nome hanno un nome associato che risulta essere il primo campo della struttura.

Esempio: `IDL > input= { image_struct, filename: '', image: bytarr(3, 256, 256) }`

input è una variale di tipo struttura, *image_struct* è il nome della struttura, *filename* è la variabile che contiene il nome del file su disco, *image* è la variabile che contiene l'immagine.

Viceversa le strutture senza nome non hanno alcun nome simbolico associato.

Le strutture con nome permettono di definire una variabile facendo riferimento al nome della struttura; per esempio l'istruzione

```
IDL > input2 = { image_struct }
```

definisce una variabile di tipo *image_struct*, i cui campi sono inizializzati a valori nulli, in altre parole si copia il formato non il contenuto.
L'istruzione

```
IDL > arr= replicate ( { image_struct }, 100 )
```

permette di definire un array di 100 elementi di tipo *image_struct*.

Inoltre, definita una struttura con nome, è possibile definire un'altra variabile evitando di usare i nome dei campi:

```
IDL > input3= { image_struct, nome, image }
```

In questo caso però se i tipi non corrispondono si verifica un errore.

Nota: per assegnare dei nuovi valori ai campi di una struttura, **non** si deve far uso dei *duepunti* (:) **ma** bensì del segno di *uguale*.
Esempio: `IDL > input.filename= 'pippo.tif'`

Le funzioni N_TAGS e TAG_NAMES sono utili per conoscere quanti campi possiede una certa struttura e i nome dei relativi campi della struttura.
Esempi:

```
IDL > print, N_TAGS ( input )
IDL > print, TAG_NAMES ( input )
```

Per conoscere il nome di una struttura si utilizza la funzione TAG_NAMES specificando il paramentro /STRUCTURE_NAME.
Esempio: `IDL > print , TAG_NAMES ( input , /STRUCTURE_NAME )`

Vettori e Matrici (Array)

Un array è un'aggregazione di dati dello stesso tipo. A seconda di come sono logicamente (e non fisicamente) organizzati i dati si parla di matrici unidimensionali (vettori), di matrici bidimensionali, ecc. Per IDL i vettori sono matrici ad una dimensione, e gli array possono avere al più 8 dimensioni. Per definire un array si può far uso delle seguenti funzioni:

BytArr	Crea un array di elementi di tipo *byte*
	`IDL > a= bytarr (10, 10)` | crea un array di 10x10 elementi di tipo *byte*
IntArr	Crea un array di elementi di tipo *integer*
	`IDL > a= intarr (100)` | crea un vettore di 100 elementi di tipo *integer*
LonArr	Crea un array di elementi di tipo *long integer*
	`IDL > a= lonarr(10,10,10)` | crea un array di 10x10x10 *long integer*
FltArr	Crea un array di elementi di tipo *floating point* a singola precisione
DblArr	Crea un array di elementi di tipo *floating point* a doppia precisione
ComplexArr	Crea un array di elementi di tipo *complex* a singola precisione
DComplexArr	Crea un array di elementi di tipo *complex* a doppia precisione
StrArr	Crea un array di elementi di tipo *string*

Inizializzazione di array

Le funzione *BytArr*, *LonArr*, *FltArr*, ecc., permettono di definire array di dimensioni specificate i cui elementi sono posti uguali a zero. Se si vogliono inizializzare gli elementi di un array ad un valore diverso, è possibile sommare agli elementi dell'array un nuovo valore.

Consideriamo per esempio la seguente definizione:
`IDL > arr= fltarr (10, 10)` | gli elementi di *arr* hanno tutti valore 0.0

L'istruzione seguente permette di assegnare agli elementi di arr un nuovo valore o, in altre parole, inizializzare l'array:
`IDL > arr= arr + 10.0` | ora tutti gli elementi di *arr* hanno valore 10.0

Attenzione: per quanto detto in precedenza, l'istruzione:
```
IDL > arr= 10.0
```
non inizializza l'array: dopo quest'istruzione *arr* è un floating point che vale 10.0.

Un modo alternativo per definire e inizializzare un array è far uso della funzione *make_array*:
```
IDL > a= make_array(10, /int)          | a è un vettore di 10 integer
IDL > a= make_array(10, 10, 10, /int) | a è un array di 10x10x10 integer
IDL > a= make_array(1000, /float)      | a è un array di 1000 floating point
IDL > a= make_array(100, /string)      | a è un array di 100 stringhe
```

Un array lo si può definire anche in modo diretto.
Esempi:
```
IDL > filenames = ['fluss00.jpg', 'flus01.jpg', 'flus02.jpg']
IDL > sobelx= [ [1., 0., 1.], [2., 0., 2.], [1., 0., 1.] ]
```

Esistono poi particolari funzioni che permettono di definire e inizializzare in modo particolare un array. In particolare la funzione *INDGEN* permette di definire un array i cui elementi hanno valore 0, 1, 2, 3, …

Esempio:
`IDL > a= INDGEN(5)` | *a* è un array di 5 elementi, precisamente *a* = [0, 1, 2, 3, 4]

In modo analogo alla funzione *INDGEN* si possono usare le funzioni *BINDGEN*, *LINDGEN*,

FINDGEN, *DINDGEN*, *CINDGEN*, *DCINDGEN* e *SINDGEN* che definiscono rispettivamente un array di byte, long, floating, point, double, complex, double complex e string.

Esercizio 3: Verificare il risultato, sulla finestra di log, delle istruzioni seguenti:

```
IDL > a= sindgen(5)
IDL > help, a
IDL > print, a
```

Utilizzo di array

IDL permette di utilizzate un array come argomento di una funzione matematica.

Esempio:

```
IDL > x= findgen(100)/99.*2.*!PI        | definisco 100 punti di campionamento
IDL > sin_func= SIN(x)                  | campiono la funzione sin
IDL > PLOT, x, sin_func                 | disegna la funzione sin(x) con x: 0 .. 2π
```

Inoltre, il linguaggio permette un ***indirizzamento multiplo*** degli elementi di un array.

Esempi:

```
IDL > arr= fltarr(10, 20)     | definisco una matrice di 10x20 floating point
IDL > arr[*, *]= 10.0         | tutti gli elementi di arr sono inizializzati al valore 10.0
IDL > arr[0 : 5, *]= 20.0     | tutti gli elementi corrispondenti alle prime 6 colonne della
matrice
                              | arr sono posti uguali a 20.0

IDL > a= bytarr (100, 200)
IDL > b= bytarr(200, 400)
IDL > b[13, 24]= a            | copio a in b a partire dalla posizione (13, 24)
IDL > b[13, 24]=
       a[137:142, 185:195]    | copio una porzione di a in b a partire dalla posizione (13, 24)
```

Un array può anche essere utilizzato per indirizzare gli elementi di un altro array.

Esempio:

```
IDL > one = [6, 5, 8, 4, 3]
IDL > two = [0, 2, 4, 1]
IDL > print, one[two]         | l'output è il seguente: 6  8  3  5
```

Una funzione molto utilizzata per l'indirizzamento multiplo degli elementi di un array è la funzione ***where***, la quale permette di indirizzare gli elementi dell'array che verificano certe condizioni; se nessun elemento dell'array verifica le condizioni specificate, la funzione ritorna –1.

Esempio:

```
IDL > data= randomu(X, 1000)          |definisco 1000 valori in modo random
IDL > indexes= where ( data gt 0.3 and data lt 0.6 )
IDL > data[indexes]= 1
```

Nota

: Nel caso in cui la funzione *where* ritorni –1, l'istruzione precedente induce un errore a run time; per evitare quest'errore si opera come segue:

```
IDL > indexes= where (data gt 0.34 and data lt 0.6, count)
IDL > if ( count GT 0 ) then data[indexes]= 1
```

Conversioni automatiche con array

Consideriamo ora le conversioni automatiche che IDL compie nella valutazioni di espressioni che contengono dati di tipo diverso e, in particolare, array di tipo diverso.

Esempi:

```
IDL> arr=bytarr(10)
IDL> help, arr
ARR             BYTE      = Array[10]
IDL> print, arr
  0  0  0  0  0  0  0  0  0  0
IDL> arr[*]=10.6
IDL> help, arr
ARR             BYTE      = Array[10]
IDL> print, arr
 10  10  10  10  10  10  10  10  10  10
IDL> arr[*]= complex(9,5)
IDL> help, arr
ARR             BYTE      = Array[10]
IDL> print, arr
  9  9  9  9  9  9  9  9  9  9
IDL> arr[*]= 300                          | 300 è un integer (!)
IDL> help, arr
ARR             BYTE      = Array[10]
IDL> print, arr
 44  44  44  44  44  44  44  44  44  44
IDL> arr[*]= '100 Euro'                   | '100 Euro' è una stringa (!)
IDL> help, arr
ARR             BYTE      = Array[10]
IDL> print, arr
100 100 100 100 100 100 100 100 100 100
```

Puntatori

Una variabile di tipo puntatore è un riferimento ad un'altra variabile. Per definire una variabile di tipo puntatore si utilizza la funzione *PTR_NEW*. L'asterisco (*) preposto alla variabile puntatore permette di riferirsi alla variabile puntata.

Esempi:

```
IDL > a= PTR_NEW(2.0)        | a punta ad una variabile floating point che vale 2.0
IDL > *a= *a * 3.            | dopo quest'operazione a punta ad una variabile floating point
                             | che vale 6.0

IDL > x= PTR_NEW()           | x è un puntatore che vale NULL
```

Le funzioni *PTRARR, PTR_VALID* e la procedura *PTR_FREE* permettono rispettivamente di creare un array di puntatori a *NULL*, di verificare se la variabile puntatore vale *NULL* e di distruggere la variabile puntata, liberando la memoria allocata.

Esempio:

```
IDL > x= PTRARR(5)
IDL > x[0]= PTR_NEW(8)
IDL > PTR_FREE, x[0]
```

Quando si lavora con i puntatori, è importante ricordarsi di liberare la memoria occupata.

Esempio:
```
IDL > a= ptr_new(3.0)
IDL > a= 4.5                    | in questo modo si spreca della memoria heap
```

Nota: per controllare quanta memoria heap è in uso si usa la procedura *help* specificando la parola chiave */memory*.

La parola chiave */NO_COPY*, specificata come parametro alla funzione *PTR_NEW*, permette di costruire un puntatore che punta ad una variabile, evitando di allocare nuova memoria; la variabile passata come parametro alla funzione *PTR_NEW*, assume tipo *UNDEFINED* (non può essere più utilizzata).

Esempio:
```
IDL > sInfo= { a: 0L, b: 'stringa' }
IDL > hInfo= PTR_NEW(sInfo, /NO_COPY)      | dopo quest'istruzione, non si può più far
                                           | riferimento alla variabile sInfo
IDL > (*hInfo).a= 1L
IDL > (*hInfo).b= 'nuova_stringa'
```

Espressioni e operatori

Variabili e costanti si possono combinare in espressioni usando operatori e funzioni. IDL permette di costruire espressioni semplici ma estremamente potenti, per esempio contenenti array.

Esempio:
```
IDL > mio_array= make_array(100, 100, /int, VALUE= 5)
IDL > mio_array= mio_array * 2
```

Oltre agli usuali operatori addizione (+), sottrazione (−), moltiplicazione (*), divisione intera e floating point (/), di modulo (MOD), di elevamento a potenza (^), di relazione (LT, LE, GE, GT, EQ, NE), booleani (AND, OR, NOT, XOR), esistono degli operatori particolari:

- moltiplicazione matriciale righe−per−colonne (##):moltiplica le righe del primo array per le colonne del secondo array
- moltiplicazione matriciale colonne−per−righe (#):moltiplica le colonne del primo array per le righe del secondo array (è utile per costruire maschere di convoluzione simmetriche)
- di minimo (<): ritorna il minimo tra due valori
- di massimo (>): ritorna il minimo tra due valori

Esempio:
```
IDL > print, 2 < 4              | stampa 2
IDL > print, 3 / 2              | stampa 1
IDL > print, 3 / 2.             | stampa 1.5
IDL > arr= arr < 100            | assegna a tutti gli elementi dell'array arr che superano il
                                | valore 100 il valore 100
IDL > print, 3 / 2.             | stampa 1.5
```

Esempio:

```
IDL> x= [1, 2, 1]               | # è utile per costruire maschere di convoluzione
simmetriche
```

```
IDL> m= x # x
IDL> print, m
       1       2       1
       2       4       2
       1       2       1
IDL> x= [1, 2, 4, 2, 1]
IDL> print, x # x
       1       2       4       2       1
       2       4       8       4       2
       4       8      16       8       4
       2       4       8       4       2
       1       2       4       2       1
```

Gli operatori di relazione possono essere utilizzati con gli array per costruire semplici ma potenti espressioni, in questo caso la relazione è valutata su tutti gli elementi dell'array.

Esempi:

```
IDL > a= randomu(X,1000)*200
IDL > b= a GT 100              | ritorna un array il cui elemento di posizione p vale 1 se a[p]
                               | è maggiore di 100, 0 altrimenti
IDL > c= a * (a GT 100)        | quest'espressione permette di "mascherare" gli elementi di
                               | a che sono minori o uguali a 100: essi sono posti a 0
IDL > print, TOTAL(a GT 100)   | stampa il numero di elementi di a maggiori di 100
```

Visualizzazione di dati

IDL mette a disposizione diverse procedure e funzioni che permettono di rappresentare a video dei dati (tipicamente degli array mono/bi/tridimensionali) in una qualche forma evocativa, per una facile interpretazione.

La funzione *PLOT* permette di disegnare un array di valori che sono interpretati come il campionamento di una certa funzione.

Esempio:
```
IDL > dati= [100, 200, 150, 175, 200, 250, 230, 200, 160, 180]
IDL > anni= 1970 + INDGEN(10) * 5
IDL > PLOT, anni, dati, XTITLE= 'Anni', YTITLE= 'Produzione'
```

Esempio:
```
IDL > x= findgen(100)
IDL > func= sin(x/5)/exp(x/35)
IDL > PLOT, func, XTITLE= 'Andamento temporale'
IDL > PLOT, x/99.*6.-3., func          | specifico l'etichettatura dell'asse delle
                                       | ascisse, ma IDL "aggiusta il tiro"
```

La funzione *PLOT* può accettare molti parametri (parole chiave) (si veda in proposito l'aiuto in linea), tra cui *XRANGE* e *YRANGE* che permettono di specificare gli intervalli di escursione degli assi.

Esempio: `IDL > PLOT, anni, dati, XRANGE= [1980, 2000], YRANGE=[90, 270]`

Se si desidera sovrapporre ad una serie di dati una seconda serie di dati (magari utilizzando un colore diverso) si deve far uso della funzione *OPLOT*.

Esempio: `IDL > OPLOT, anni, dati * 1.1, COLOR= 200`

Nota: La funzione *OPLOT* non accette le parole chiave *XRANGE* e *YRANGE*, quindi la prima serie di dati (quella disegnata mediante la funzione *PLOT*) dev'essere la più "ampia", oppure si specificano, nella funzione *PLOT*, i parametri *XRANGE* e *YRANGE* corrispondenti agli intervalli più ampi.

Per scrivere un testo a fianco di un grafico può essere utile la funzione *XYOUTS*.

Esempio:

```
IDL > XYOUTS, 100, 150, "Stringa",
       color= 200, CHARSIZE= 3, /device        | stampa a posizione (100,150); l'origine si
                                               | trova in basso a sinistra
```

La funzione *PLOT* connette i punti della serie di dati con segmenti di retta.

Esempio: `IDL > plot, [0, 1, 0], yrange=[-1, 2]`

Per tracciare segnali particolare quali p.e. le onde quadre, è necessario specificare la parola chiave *PSYM= 10*, in questo modo la funzione *PLOT* opera in modalità "istogramma".

Esempio: `IDL > oplot, [0, 1, 0], psym=10`

Nota: altre parole chiavi che possono risultare utili sono *MAX_VALUE* e *MIN_VALUE* che permettono di annullare i valori superiori o inferiori ad una certa costante specificata; in questo modo, se i dati contengono "elementi spuri", è possibile evitare di visualizzare questi picchi.

IDL mette a disposizione funzioni anche per la visualizzazione di dati bidimensionali. In particolare la funzione *SURFACE* interpreta gli elementi di un array 2D come i valori di elevazione di una funzione $f : \Re^2 \rightarrow \Re$ e disegna una superficie connettendo con segmenti i punti vicini (evitando di tracciare i segmenti nascosti).

Esempio:

```
IDL > XY= SHIFT(DIST(50), 25, 25)      | definiamo i punti di campionamento
IDL > Z= EXP(-(XY/15)^2)               | campioniamo una gaussiana 2D
IDL > SURFACE, Z
```

Note:
IDL mette anche a disposizione una procedura per il rendering di superfici (*SHADE_SURF*), ed inoltre permette di decidere la posizione della sorgente di luce incidente la superficie (*SET_SHADING*).
E' anche disponibile un programma, scritto in IDL, di nome *XSURFACE* che permette di visualizzare e manipolare in modo interattivo i parametri di visualizzazione di una superficie.
Esempio:

```
IDL > XY= SHIFT(DIST(50), 20, 15)      | "interessante" dataset
IDL > XSURFACE, EXP(-(XY/15)^2)
```

Inoltre è possibile lavorare con array 3D i cui elementi rappresentano delle densità (si pensi ad una serie di lastre di RMN), per esempio per visualizzare i contorni 3D di regioni omogenee.

I/O e visualizzazione di immagini

Una delle capacità più interessanti di IDL riguarda la lettura di file grafici e la visualizzazione delle immagini corrispondenti, oltre che l'elaborazione di queste immagini. In pratica esistono delle subroutine che permettono di caricare in memoria (in una variabile) le immagini memorizzate in file *.BMP*, *.TIF* (non compresso) e altri tipi di file. E' anche possibile salvare su disco delle immagini utilizzando analoghe subroutine.

Esempi:
```
IDL > img= read_tiff('nome_file.tif')
IDL > read_jpeg, 'nome_file.jpg', img2
```

Un'immagine digitale concettualmente è un array bidimensionale (2D) di elementi che rappresentano un livello di grigio o un colore. Più in dettaglio un'immagine a livelli di grigio è rappresentata come un array bidimensionale di *byte*, cioè ogni elemento dell'array può assumere un valore compreso tra 0 e 255. Allo 0 corrisponde il nero mentre al 255 corrisponde il bianco. Un'immagine di questo tipo è direttamente visualizzabile mediante la funzione *TV*.

Esempio: `IDL > TV, img_grey`

La funzione *TV* apre una finestra (*window*) sul desktop e disegna in essa l'immagine. Le dimensioni della finestra generata dalla funzione *TV*, in generale, non corrispondono alle dimensione dell'immagine da visualizzare. A questo riguardo si può utilizzare la funzione *window*, e operare come segue:

```
IDL > dim= size(img_grey)
IDL > dimx= dim[1]
IDL > dimy= dim[2]
IDL > window, /free, xsize= dimx, ysize= dimy
IDL > tv, img_grey
```

La parola riservata */free* passata come parametro alla funzione *window* permette di generare una nuova finestra, mentre i parametri *xsize* e *ysize* specificano le dimensioni della finestra. Se è presente sul desktop più di una window, la funzione *TV* opera sull'ultima, mentre se non ve ne nessuna ne genera una nuova.

Nota: la funzione *size* permette di ottenere le dimensioni di un array: ritorna un array di cui il primo elemento specifica il numero di dimensioni dell'array, e gli elementi successivi le rispettive dimensioni. *Esempio*: sia ***x*** un array di tipo *bytarr(3,100,250)*, allora *size(**x**)* ritorna un array di 4 elementi; precisamente [0]= 3, [1]=3, [2]= 100, e [3]= 250.

Nota: per la gestione di più finestre si usa la funzione *WSET*, che permette di specificare quale finestra dovrà essere considerata per le future visualizzazioni.

Veniamo ora alle immagini a colori. Un'immagine digitale a colori può essere rappresentata da un array bidimensionale o da un array tridimensionale. Nel primo caso ogni elemento dell'array è un byte che rappresenta l'indice di un colore memorizzato in una tabella denominata *color table*, nel secondo caso, ad ogni pixel dell'immagine sono associati 3 byte, uno per ogni colore primario, secondo il modello di colore *RGB*. Nel primo caso si parla di una *profondità di colore* di 8 bit mentre nel secondo caso si parla di una *profondità di colore* di 24 bit.

Se *img_color24* è un'immagine a colori del secondo tipo per visualizzarla si usa la funzione *TV*, però specificando che si tratta di un'immagine a colori a 24 bit:

Esempio: `IDL > TV, img_color24, true = 1`

Il parametro *true = 1*, passato alla funzione *TV*, indica che l'array 3D è tale per cui il primo indice

varia tra 0 e 2 (3 dimensioni) cioè *img_color24* è del tipo *bytarr(3, dimx, dimy)*.

Viceversa se *img_color24* fosse di tipo *bytarr(dimx, 3, dimy)* allora *true* si deve porre uguale a 2, mentre se è di tipo *bytarr(dimx, dimy, 3)* allora a *true* si deve assegnare il valore 3. Comunque, in genere un'immagine a colori viene definita come di tipo *bytarr(3, dimx, dimy)*; anche le funzioni proprie di IDL per la lettura dei file grafici (*.BMP*, *.TIF*, *.JPG*, ...) ritornano l'immagine in un array di questo tipo.

Se supponiamo che la profondità di colore sia di 8 bit, si utilizza ancora la funzione *TV* ma in questo caso o non si specifica il parametro *true* oppure lo si specifica ma con valore 0.

Esempio:

```
IDL > TV, img_color, true = 0
IDL > TV, img_color
```

***Nota**:* A volte può risulate utile trasformare un'immagine a colori a 24 bit in un'immagine a colori a 8 bit, per esempio per risparmiare risorse di calcolo. A tal fine è possibile specificare il numero massimo di colori che un'immagine può avere direttamente nella funzione di caricamento dell'immagine da disco.

Esempio:

```
IDL > read_jpeg, 'dna.jpg', img, colortable, colors= 256
IDL > tvlct, colortable
IDL > window, /free, xsize= (size(img))[1], ysize= (size(img))[2]
IDL > tv, img
IDL > help, img
```

Un'altra funzione in grado di visualizzare un'immagine è la funzione *TVSCL*. Questa funzione effettua uno scalamento di valori dell'immagine prima di visualizzarla, in modo da aumentare la gamma dinamica dei colori o dei livelli di grigio dell'immagine; quindi se per esempio l'immagine era tra grigio chiaro e grigio scuro diventerà tra nero e bianco. Per il resto la funzione *TVSCL* si usa come la funzione *TV*.

Operatori di IDL di image processing

IDL ha diversi operatori implementati e direttamente utilizzabili per elaborare un'immagine.

Calcolo e visualizzazione dell'istogramma

La funzione *HISTOGRAM* ritorna l'istogramma di un'immagine.

Esempio:

```
IDL > PLOT, HISTOGRAM(img_grey)
IDL > PLOT, HISTOGRAM(img_grey), MAX_VALUE= 5000
```

Sogliatura (binarizzazione)

Per binarizzare un'immagine è possibile far uso degli operatori di comparazione EQ, LT, GT, LE, GE e NE che ritornano 1 se la relazione in cui compaiono è verificata, 0 altrimenti; se nella espressione compare un'array (un'immagine) la relazione è valutata per ogni valore dell'array.

Così il comando:

```
IDL > img_bin= img_grey GT 140
```

permette di costruire un'immagine binaria in cui ad ogni pixel con valore 1 corrisponde un pixel con valore maggiore di 140 nell'immagine *img_grey*, e analogamente ai pixel di *img_bin* che hanno valore 0 corrispondono i pixel di *img_grey* che hanno valore maggiore di 140.

Questi operatori si possono usare anche con le funzioni *TV* e *TVSCL*. Così sono lecite le istruzioni seguenti:

```
IDL > TV, (img_grey GT 140) * 255
IDL > TVSCL, img_grey GT 140
IDL > TVSCL, BYTSCL(img_grey GT 140)
```

La funzione *BYTSCL* scala i valori di un array e ritorna un array di *BYTE* i cui elementi variano tra 0 e 255. Parametri interessanti per questa funzione sono: *MIN*, *MAX* e *TOP*: tutti i valori minori di *MIN* sono posti a 0 e tutti i valori maggiori di *MAX* sono posti a *TOP*, gli altri valori sono scalati all'interno di 0 e *TOP*.

Equalizzazione

La funzione *HIST_EQUAL* effettua l'equalizzazione di un'immagine.

Esempi: `IDL > TV, HIST_EQUAL(img_grey)`

Smoothing (sfuocatura)

Un operatore di sfuocatura di un'immagine si può effettuare mediante la funzione *SMOOTH*:

```
IDL > TVSCL, SMOOTH(img_grey, 7)
```

Tale funzione può essere utilizzata anche per aumentare il contrasto di un'immagine:

```
IDL > TVSCL, FIX(img_grey − SMOOTH(img_grey, 7))
```

```
La funzione SMOOTH può essere utilizzata anche con array monodimensionali:
IDL > window, /free
IDL > PLOT, HISTOGRAM(img_grey)
IDL > window, /free
IDL > PLOT, SMOOTH(HISTOGRAM(img_grey), 7)
```

Operatori per l'estrazione dei bordi

Gli operatori di derivazione permettono di estrarre i contorni degli oggetti rappresentati nell'immagine:

```
IDL > TV, ROBERTS(img_grey)
IDL > TV, SOBEL(img_grey)
```

Nota: Si può anche elaborare solo una parte dell'immagine. Per esempio:

```
IDL > img_out= img_grey
IDL > img_out(50:100, 50:100)= SMOOTH(img_grey(50:100, 50:100), 7)
IDL > TV, img_out
```

Filtro mediano

Un comune tipo di rumore che si può presentare nelle immagini, p.e. dovuto alla trasmissione delle immagini, è denominato rumore sale−pepe; si presenta come un'alterazione casuale di pixel che assumono valori estremi di livelli di grigio. IDL implementa un tipico filtro (filtro mediano) per la riduzione di questo tipo di rumore.

Esempio (da vedere!):

```
IDL> read_jpeg, 'car.jpg', img_grey
IDL> dim= size(img_grey)
IDL> dimx= dim[1]
IDL> dimy= dim[2]
IDL> window, /free, xsize= dimx, ysize= dimy, title='Immagine originale'
IDL> TV, img_grey
IDL> punti= randomu(X, dimx*dimy*.5) * dimx * dimy
IDL> img_grey[punti]= 255
IDL> punti= randomu(X, dimx*dimy*.5) * dimx * dimy
IDL> img_grey[punti]= 0
IDL> window, /free, xsize= dimx, ysize= dimy, title='Immagine con rumore'
IDL> TV, img_grey
IDL> window, /free, xsize= dimx, ysize= dimy, title='Immagine restaurata'
IDL> TV, median(img_grey, 5)
```

Esercizio 4: Si consideri l'immagine a colori 'dna.jpg'; si supponga di voler applicare il filtro mediano a ciascun piano colore, in modo indipendente, come fare ? Si verifichi che non è possibile chiamare la funzione median specificando come parametro img_dna(i,,*) dove i è l'i−esimo piano colore. Suggerimento: si utilizzi un array 2D ausiliario in cui copiare l'i−esimo piano colore.*

Costruzione di un operatore (lineare)

Vediamo come si definisce e si usa un operatore lineare. Costruiamo per esempio l'operatore di Sobel:

```
  sobelx= [ [ 1.,  0., −1.], $
            [ 2.,  0., −2.], $
            [ 1.,  0., −1.] ]

  sobely= [ [ 1.,  2.,  1.], $
            [ 0.,  0.,  0.], $
            [−1., −2., −1.] ]

img_sobel= sqrt( convol(img, sobelx, /center)^2 + $
                 convol(img, sobely, /center)^2 )
```

Esercizio 5: Come fare per costruire le maschere di convoluzione sobelx e sobely utilizzando l'operatore # ?

Ricampionamento di un'immagine

Le funzioni *REBIN* e *CONGRID* possono risultare utile per uno scalamento di un'immagine.

La funzione *REBIN* ridimensiona un array di un fattore intero.

Esempio:

```
IDL > read_jpeg, 'car.jpg', img
IDL > window, /free, xsize= 200, ysize= 100
IDL > TV, img[100:299, 100:199]
```

```
IDL > window, /free, xsize= 2*200, ysize= 2*100
IDL > TV, REBIN(img[100:299, 100:199], 2*200, 2*100)
```

La funzione *CONGRID* ridimensiona un array di un fattore qualsiasi e permette di specificare la tecnica di estrapolazione (si veda in proposito l'help in linea).

Esempio:
```
IDL > read_jpeg, 'car.jpg', img
IDL > window, /free, xsize= 200, ysize= 100
IDL > TV, img[100:299, 100:199]
IDL > window, /free, xsize= 180, ysize= 250
IDL > TV, CONGRID(img[100:299, 100:199], 180, 250)
```

Erosione e Dilatazione

Le funzione *ERODE* e *DILATE* permettono di effettuare delle operazioni di morfologia matematica, sia sulle immagini binarie che su quelle a livelli di grigio.

Esempio:

```
IDL > read_jpeg, 'car.jpg', img
IDL > dim= size(img)
IDL > window, /free, xsize= dim[1], ysize= dim[2], title='Immagine originale'
IDL > tv, img
IDL > disk= [1, 1, 1, 1, 1]
IDL > disk= disk # disk
IDL > window, /free, xsize= dim[1], ysize= dim[2], title='Immagine dilatata'
IDL > tv, dilate(img, disk, /gray)
IDL > window, /free, xsize= dim[1], ysize= dim[2], title='Immagine erosa'
IDL > tv, erode(img, disk, /gray)
IDL > window, /free, xsize= dim[1], ysize= dim[2], title='Opening'
IDL > tv, dilate(erode(img, disk, /gray), disk, /gray)
IDL > window, /free, xsize= dim[1], ysize= dim[2], title='Closing'
IDL > tv, erode(dilate(img, disk, /gray), disk, /gray)
```

Vediamo un esempio applicato ad immagini binarie (codificate con 0 e 1). In questo caso "si erode" o "si dilata" il bianco (i punti che hanno valore 1), che può rappresentare l'oggetto o lo sfondo a seconda del caso.

Esempio:

```
IDL > read_jpeg, 'cubi.jpg', img
IDL > dim= size(img)
IDL > window, /free, xsize= dim[1], ysize= dim[2], title='Immagine originale'
IDL > tv, img
IDL > img_bin= img GT 100
IDL > window, /free, xsize= dim[1], ysize= dim[2], title='Immagine sogliata'
IDL > tv, img_bin
IDL > disk= [1, 1, 1, 1, 1]
IDL > disk= disk # disk
IDL > window, /free, xsize= dim[1], ysize= dim[2], title='Immagine dilatata'
IDL > tvscl, dilate(img_bin, disk)
IDL > window, /free, xsize= dim[1], ysize= dim[2], title='Immagine erosa'
IDL > tvscl, erode(img_bin, disk)
IDL > window, /free, xsize= dim[1], ysize= dim[2], title='Opening'
IDL > tvscl, dilate(erode(img_bin, disk), disk)
IDL > window, /free, xsize= dim[1], ysize= dim[2], title='Closing'
IDL > tvscl, erode(dilate(img_bin, disk), disk)
```

Costruzione di programmi

I programmi IDL sono memorizzati in file di testo che, per convenzione, hanno estensione *.pro*.

Commenti

I commenti in IDL devono seguire il carattere ; (punto−e−virgola).

Istruzioni su più righe

Il carattere $ (dollar) permette di spezzare un'unica istruzione su più righe.
Esempio:

```
print,    struct.nome, $
          struct.cognome, $
          struct.indirizzo
```

Più istruzioni su di una riga

Il carattere & (ampersand) permette di scrivere diverse istruzioni su di una riga.
Esempio:

```
wset, windows1st  &  tv, image, true= 1
```

Istruzioni del linguaggio

Istruzioni di selezione

- ***If then else***

L'istruzione *if* è un'istruzione di selezione che permette di valutare un'espressione e di scegliere quali istruzioni far interpretare ed eseguire all'interprete.

La sintassi completa dell'istruzione è la seguente:

```
if <espressione> then <istruzioni> [else <istruzioni> ]
```

Esempio 1:

```
if ( nome EQ 'Lucia' ) $
    then print, 'Sorella' $
    else print, 'Fratello'
```

La sezione *else* non è obbligatoria.

Esempio 2:

```
if ( nome EQ 'Lucia' ) then begin
    print, 'Sorella'
endif else begin
    print, 'Fratello'
endelse
```

- ***Case***

L'istruzione *case* è un'istruzione di selezione multipla. Ci sono due schemi (duali) di utilizzo dell'istruzione; il primo prevede di specificare la variabile da verificare e di seguito i valori che può assumere, il secondo prevede di specificare una serie di espressioni, e l'espressione che assume il valore di verità 1 individua l'istruzione (o del blocco di istruzioni) da eseguire.

Esempio 1:

```
case ( nome ) of
```

```
    "Lucia" : print, 'Lucia'
    "Mara" : begin
        print, 'Mara'
    end
    else : print, 'Non conosco'
endcase
```

La sezione *else* non è obbligatoria, se però non è presente e la variabile da valutare (*nome* nell'esempio precedente) non corrisponde a nessuna voce allora l'interprete segnalerà un errore di run time.

Esempio 2:

```
case ( 1 ) of
    (nome EQ 'Lucia') : print, 'Lucia'
    (nome EQ 'Mara') : print, 'Mara'
    else: print, 'Non conosco'
endcase
```

Nello schema precedente, se più di una espressione ha valore di verità 1, solo il blocco di istruzioni corrispondente alla prima espressione verificata, incontrata dall'interprete, viene eseguito. Anche in questo caso, se nessuna espressione ha valore di verità 1, si verifica un errore di run time.

Istruzioni di sequenza

I programmi IDL che non fanno uso delle interfacce grafiche (*widget*) sono eseguiti un'istruzione alla volta, dalla prima all'ultima istruzione, cioè in sequenza. Ci sono delle istruzioni che permettono di alterare il normale flusso di esecuzione.

- ***Label − Goto***

Le *label* permettono di identificare un punto di un programma al quale si può "saltare" mediante l'istruzione *GOTO*. Le regole per definire nomi di label validi sono le stesse di quelle per i nomi delle variabili.

Esempio:

```
pro
    …
    if ( errore ) $
        then goto, end_of_proc
    …
    end_of_proc:
end
```

Nota: Le label sono spesso utilizzate per la gestione degli errori (si vedano in proposito, sull'help in linea, le procedure ON_ERROR e ON_IOERROR)

- ***STOP e .CONT***

Quando l'interprete incontra l'istruzione *STOP*, sospende l'esecuzione del programma; l'utente può accedere alle variabili e può immettere istruzioni al prompt dei comandi di IDL. Se si immette il comando *.CONT* (nota il punto preposto al comando) al prompt dei comandi, l'interprete riprende l'esecuzione del programma.

Istruzioni di iterazione

- ***For next***

L'istruzione *FOR* permette di eseguire più volte la stessa istruzione (o blocco di istruzioni), fino a quando la variabile contatore non supera in valore la variabile limite; tale condizione è verificata prima di ciascuna iterazione (anche della prima).

La sintassi completa dell'istruzione è la seguente:

for var_cont = <valore> , var_lim [, var_incr] do <istruzioni>

Se non si esplicita l'incremento che deve subire la variabile contatore ad ogni iterazione, questa viene incrementata di 1.

Esempio:

```
for i = 0 , 50 do begin
        print, i
endfor
```

Se la variabile contatore e la variabile limite sono di tipo differente, IDL cambia sembre il tipo **della variabile limite**, e questo può comportare dei problemi.

Esempio:

```
for i = 25000 , 50000 do begin      | in questo caso il ciclo non viene eseguito
        print, i                    | infatti fix(50000) = −15536
endfor
```

La scrittura corretta è la seguente:

```
for i = 25000L , 50000 do begin
                print, i
endfor
```

Nota: il tipo della variabile contatore non può cambiare all'interno del ciclo di iterazione.

Un altro problema può derivare dall'uso combinato dei caratteri $ e &.

Esempio:

```
for i = 0 , 10 do $
        print, i   &   print, i/2          | la seconda print non è eseguita ad ogni
iterazione !
```

Esercizio 6: Scrivere un programma per il calcolo della funzione istogramma data un'immagine a livelli di grigio.

La variabili contatore e limite possono anche essere di tipo floating point con incrementi frazionari.

Esempio:

```
for x= 1.5 , 10.5 do begin          | x assume i valori 1.5, 2.5, 3.5, 4.5, 5.5, ecc
            …
endfor
```

Attenzione però:

```
for x= 0 , 1, 0.1 do begin          | il ciclo non viene eseguito perché 0 è integer
            …
```

```
endfor
```

Esercizio 7: Correggere il precedente esempio.

- ***While do***

L'istruzione *While do* permette di eseguire un'istruzione o un blocco di istruzioni finché una certa condizione specificata non rimane vera, quando diviene false si esce dal ciclo.

Esempio:

```
i= 10
while ( i GE 0 ) do begin
        …
        i= i − 1
endwhile
```

- ***Repeat until***

L'istruzione *Repeat until* è simile all'istruzione *While do*, solamente che qui il ciclo continua finché la condizione specificata non rimane falsa, quando diviene vera si termina.

Esempio:

```
repeat begin
        ..
        num= num + 2
endrep until ( num GT 20 )
```

Procedure e funzioni

Le funzioni e le procedure sono pezzi di programma che svolgono compiti ben delineati. Per chiamare una procedura o una funzione si usa il nome della subroutine.

Una procedura inizia con la parola riservata *PRO* e termina con la parola riservata *END*.

Esempio:

```
PRO Hello
        print, 'Ciao Mondo!'
END
```

Una funzione inizia con la parola riservata *FUNCTION* e termina con la parola riservata *END*; inoltre, prima del termine della funzione deve essere utilizzata la parola riservata *RETURN*.

Esempio:

```
FUNCTION Hello_func
        RETURN, 'Ciao Mondo!'
END
```

Per convenzione i file di testo che contengono codice IDL hanno estensione *.pro*. Questo non è richiesto ma è necessario per una compilazione automatica delle procedure e delle funzioni. Più precisamente, quando l'interprete di IDL incontra una chiamata ad una procedura o ad una funzione che non è stata definita nello stesso modulo in cui è posta la chiamata alla subroutine, allora IDL cerca un file nella directory corrente (e in quelle specificate nella variabile di sistema *!PATH*) con estensione *.pro* e di nome uguale al nome della subroutine chiamata. Se IDL trova il file, lo compila ed esegue autonomamente ed in modo automatico la subroutine.

Questo meccanismo di compilazione automatica sembra più intelligente di quanto in realtà non sia; sarebbe più utile poter definire un file di progetto contenente tutti i moduli dell'applicazione che si sta costruendo (come è possibile con ambienti di sviluppo più evoluti). Viceversa, per evitare "problemi", o si pone ogni subroutine in un file con nome uguale a quello della subroutine e con estensione *.pro*, oppure è necessario fare qualche "salto mortale" per far vedere tutte le subroutine a IDL (in particolare quando "A" chiama "B" che chiama "C" che a sua volta chiama "A"…). A complicare le cose, IDL non compila i moduli corrispondenti alle subroutine chiamate in relazione all'ordine di chiamata ma bensì, in relazione all'ordine alfabetico delle subroutine.

Le cose sono poi complicate ulteriormente dal fatto che la versione 5.0 di IDL si porta dietro un retaggio delle versioni precedenti, in relazione al modo con cui indirizzare gli elementi di un array. Infatti, è lecito indirizzare gli elementi di un array sia utilizzando le parentesi *tonde* sia utilizzando le parentesi *quadrate*.
Ora, supponiamo di aver definito in un modulo (=in un file di testo) una certa funzione *pippo* che accetta un parametro

```
function pippo, a
    …
end
```

e di voler chiamare la funzione da un altro modulo

```
…
ret= pippo ( 9 )
…
```

in questo caso, se la funzione pippo non è già stata compilata e inoltre non esiste un file di nome uguale al nome della function, IDL crede che noi ci si voglia riferire all'elemento 9 dell'array pippo, array che però non è stato definito e, quindi, segnala un errore.
Per evitare questo problema si deve dichiarare la funzione pippo nel modulo in cui è presenta la chiamata alla subroutine, con il l'istruzione seguente:

```
forward_function pippo, a
```

Passaggio di parametri

E' possibile passare sia alle funzioni sia alle procedure dei parametri. I parametri della subroutine seguono il nome della subroutine e sono separati da una *virgola* (,).

Esempio:

```
PRO Stampa, str
        print, str
END

FUNCTION Somma, a , b      ; a e b sono i parametri formali
        RETURN, a + b
END
…
Stampa, 'Ciao Mondo!'
result = mySum ( 10 , 14 )     ; 10 e 14 sono i parametri attuali
                               ; i parametri delle funzioni sono racchiusi da parentesi tonde
…
```

I parametri delle subroutine sono *posizionali* o di tipo *keyword* (di chiave).
I parametri posizionali sono quelli classici, specificati dalla loro posizione nella lista dei parametri.

Esempio:

```
function Somma, a, b
…
ret= Somma (23, 30)     ; chiamata alla funzione
…
```

I parametri di tipo *keyword* sono specificati da parole chiave.

Esempio:

```
function Somma, DATO1= a, DATO2= b
…
ret= Somma (DATO1= 23, DATO2= 34)     ; chiamata alla funzione
…
```

Non è necessario specificare sempre tutti i parametri di una subroutine, siano essi posizionali o di tipo keyword. Per determinare se e quanti parametri posizionali sono specificati si usa la funzione *N_Params()*:

```
function Somma, a, b, c, d
    case ( N_Params() ) of
        2: RETURN, a + b
        3: RETURN, a + b + c
        4: RETURN, a + b + c + d
    endcase
end
```

Se si fa riferimento ad un parametro non specificato, si otterrà un errore. Inoltre, se non si specificano tutti i parametri posizionali, si assume che siano stati specificati i primi *N_Params()*.

Per determinare se e quanti parametri di tipo *keyword* sono specificati si usa la funzione *N_Elements*:

```
function Somma, DATO1= a, DATO2= b, DATO3= c
    if ( N_Elements(a) EQ 0 ) then a= 0
    if ( N_Elements(b) EQ 0 ) then b= 0
    if ( N_Elements(c) EQ 0 ) then c= 0
    RETURN, a + b + c
end
```

Nota: le variabili sono sempre passate *per riferimento (by reference)*.

Variabili e Costanti globali (common)

IDL permette di definire “blocchi di variabili comuni”, specificando delle sezioni *common*.

Esempio:

```
pro alpha
common, VAR1, a, b, c
    a= 0L
    b= ‘pippo’
    c= 80.0
end
```

Per riferirsi a queste variabili si deve dichiarare l’uso della sezione *common VAR1*:

Esempio:

```
pro beta
common, VAR1
    …
end
```

Studio di alcuni semplici programmi

Vediamo ora alcuni semplici programmi.

Conversione tra spazi colore

```
pro RGB2HSV, img_rgb
  ; accetta in ingresso un'immagine a colori (rgb)
      dim= size(img_rgb)
      img_hsv= make_array(3, dim[2], dim[3], /FLOAT)
      color_convert, img_rgb(0,*,*),img_rgb(1,*,*),img_rgb(2,*,*), $
                     hh, ss, vv,/RGB_HSV
      img_hsv(0, *, *)=hh
      img_hsv(1, *, *)=ss
      img_hsv(2, *, *)=vv
      window, /free, XSIZE= dim[2], YSIZE= dim[3], TITLE= '[TRUE COLORS RGB]'
      tv, img_rgb, TRUE= 1
      window, /free, XSIZE= dim[2], YSIZE= dim[3], TITLE='[Hue]'
      tvscl, img_hsv[0, *, *]
      window, /free, XSIZE= dim[2], YSIZE= dim[3], TITLE='[Saturation]'
      tvscl, img_hsv[1, *, *]
      window, /free, XSIZE= dim[2], YSIZE= dim[3], TITLE='[Value]'
      tvscl, img_hsv[2, *, *]
end
```

Conversione di un'immagine binaria {0,1} in immagine binaria {–1,1}

```
function converti, img_in
   ; accetta in ingresso un'immagine binaria codificata con i valori 0 e 1
   ; ritorna in uscita l'immagine binaria codificata con i valori –1 e 1
      dim= size(img_in) & dimx= dim[1] & dimy= dim[2]
      img_out= make_array(dimx, dimy, /int, VALUE= 1)
      indexes= where(img_in EQ 0, count)
      if ( count GT 0 ) then img_out[indexes]= -1     ;nota l'uso di count
      return, img_out
end
```

Esercizio 8: Come visualizzare l'immagine *img_out* ?

Visualizzazione di un'onda quadra come sovrapposizioni di funzioni coseno

```
pro quadra, A, periodo, nCos
      a0= float(A)
      T= periodo/4.
      ncamp= 100.
      quadra= fltarr(ncamp)
      quadra(0 : fix(ncamp/8.))= a0
      quadra(fix(ncamp/8.*3) : fix(ncamp/8.*5))= a0
      quadra(fix(ncamp/8.*7.) : ncamp-1)= a0
      func= fltarr(ncamp)
      x= findgen(ncamp)/(ncamp-1)*2.*periodo-periodo

      func(*)= a0/2. ;primo termine
      for i= 0., nCos-1 do begin
        j= 2. * i + 1.
        aj= ((2.*A)/(!PI*j))*sin(j*!PI/2.)
        func(*)= func(*) + aj * cos((j*(2.*!PI)/periodo)*x)
      endfor
      PLOT, x, quadra, PSYM= 10, YRANGE= [-1,a0+1]
      OPLOT, x, func, color=200
end
```

Programmazione di interfacce grafiche

In genere un'applicazione è dotata di una interfaccia grafica, per esempio costituita da un menù, dei bottoni, dialoghi e altro.

Un programma che fa uso di questi elementi, denominati *widgets*, non è un programma "sequenziale" (un'istruzione dopo l'altra), ma un programma "ad eventi", e gli eventi sono generati in risposta al comportamento dell'utente del programma. In genere non ci si deve preoccupare di generare gli eventi, ma solo di programmare il modo in cui gestirli. Per dare l'idea di un evento si pensi all'evento generato in risposta alla pressione di un tasto del mouse da parte dell'utente.

I programmi guidati dagli eventi sono scritti, almeno a basso livello, in modo tale per cui spendono gran parte del loro tempo in uno stato di attesa, attesa di un qualche evento da gestire.

Ciò si traduce in pratica nel dover scrivere una o più procedure di gestione degli eventi, che non saranno mai chiamate direttamente dall'interno di altre procedure o funzioni, ma piuttosto sono attivate da IDL. Quando un evento è generato, il programma si preoccupa di gestirlo, cioè "fa qualcosa", e poi torna subito in uno stato di attesa (del prossimo evento da gestire).

Gli eventi che una certa procedura di gestione può gestire sono molti: dalla pressione di un bottone grafico, alla selezione di una voce di menù, alla pressione di un tasto del mouse, agli eventi di timer e molti altri.

Vediamo un esempio di come si può costruire una semplice interfaccia con due bottoni, e come si programma la procedura di gestione dei rispettivi eventi di pressione.

```
pro gestore_eventi1, event
```

```
;
; la procedura di gestione degli eventi riceve sempre un parametro
; in ingresso (event), che caratterizza l'evento da gestire
;
    WIDGET_CONTROL, event.ID, GET_VALUE= valore
    case valore of
        'Bottone Uno': print, 'Bottone Uno'
        'Bottone Due': print, 'Bottone Due'
    endcase
end

pro primo
    base= WIDGET_BASE(column= 1, title= 'Primo programma')
      ; base e' la finestra contenitore
      bottoneUno= WIDGET_BUTTON(base, value= 'Bottone Uno', xsize= 300)
      bottonoDue= WIDGET_BUTTON(base, value= 'Bottone Due', xsize= 300)

    WIDGET_CONTROL, base, /REALIZE
      ; questo e' il comando che costruisce la finestra 'base'
    XMANAGER, 'primo programma', base, event= 'gestore_eventi1'
end
```

La procedura *XMANAGER* informa IDL su quale sia la procedura di gestione degli eventi.

Molto spesso è conveniente riferirsi alle widget con dei nomi mnemonici; per far ciò si può operare come segue:

```
pro gestore_eventi2, event
    WIDGET_CONTROL, event.ID, GET_UVALUE= valore
      ; si noti l'uso di GET_UVALUE al posto di GET_VALUE
    case valore of
        '1': print, 'Bottone Uno'
        '2': print, 'Bottone Due'
    endcase
end

pro secondo
    base= WIDGET_BASE(column= 1, title= 'Primo programma')
    bottoneUno= WIDGET_BUTTON(base, value= 'Bottone Uno', $
                uvalue='1', xsize= 300)
      ; il parametro UVALUE permette di assegnare un nome alla widget
    bottonoDue= WIDGET_BUTTON(base, value= 'Bottone Due', $
                uvalue='2', xsize= 300)
    WIDGET_CONTROL, base, /REALIZE
    XMANAGER, 'secondo programma', base, event= 'gestore_eventi2'
end
```

Vediamo ora un esempio più complesso di programmazione di interfacce grafiche; in particolare utilizzeremo una struttura in cui memorizzare delle informazioni a cui si deve accedere dalla procedura di gestione degli eventi.

```
pro info_event, event
    WIDGET_CONTROL, event.top, /DESTROY
end

pro gestore_eventi, event
    WIDGET_CONTROL, event.ID, GET_UVALUE= valore
    WIDGET_CONTROL, event.top, GET_UVALUE= sInfo, /NO_COPY
    case valore of
        'LOAD': begin
          img= bytarr(400,400)
          OPENR, 1, 'image.dat'
          READU, 1, img
          CLOSE, 1
          wset, sInfo.wAreaID
          tv, img
        end
        'SAVE': begin
          OPENW, 1, 'image.dat'
          img= TVRD()
```

```
          WRITEU, 1, img
          CLOSE, 1
        end
        'AREA':
        'INFO': begin
          strs= ['', 'Questo è un programma che usa widgets', '']
          wBaseInfo= WIDGET_BASE(TITLE= "Informazioni Su...", $
                     GROUP_LEADER= event.top, /MODAL, /COLUMN)
            txtInfo= WIDGET_TEXT(wBaseInfo, VALUE=strs, $
                     xsize= 33, ysize= 3)
            cmdOK= WIDGET_BUTTON(wBaseInfo, value= 'OK', $
                   /ALIGN_CENTER, xsize=100, ysize= 25)
            WIDGET_CONTROL, wBaseInfo, /REALIZE
          XMANAGER, 'Informazioni Su...', wBaseInfo, $
                    event= 'info_event'
        end
        'QUIT': WIDGET_CONTROL, event.top, /DESTROY
        'Dis': begin
          wset, sInfo.wAreaID
          xy= shift(dist(50), 15, 20)
          surface, exp(-(xy/15)^2)
        end
        'Canc': begin
          wset, sInfo.wAreaID
          erase
        end
    endcase
    WIDGET_CONTROL, event.top, SET_UVALUE= sInfo, /NO_COPY
end

pro terzo
  wBase= WIDGET_BASE(TITLE= "Terzo programma", MBAR= barMenu, $
         /COLUMN)

    wFileMenu= WIDGET_BUTTON(barMenu, VALUE= 'File')
      wLoadMenu= WIDGET_BUTTON(wFileMenu, VALUE= 'Load', $
                 UVALUE= 'LOAD')
      wSaveMenu= WIDGET_BUTTON(wFileMenu, VALUE= 'Save', $
                 UVALUE= 'SAVE')
      wQuitMenu= WIDGET_BUTTON(wFileMenu, VALUE= 'Quit', $
                 UVALUE= 'QUIT')
    wInfoMenu= WIDGET_BUTTON(barMenu, VALUE= 'Informazioni')
      wAboutMenu= WIDGET_BUTTON(wInfoMenu, $
                  VALUE= 'Informazioni su..', UVALUE= 'INFO')

    wArea= WIDGET_DRAW(wBase, XSIZE= 400, YSIZE= 400, RETAIN= 2)

    wBaseButton= WIDGET_BASE(wBase, /ROW, /align_center)
      Disegna= WIDGET_BUTTON(wBaseButton, value= 'Disegna', $
               uvalue='Dis', xsize= 150)
      Cancella= WIDGET_BUTTON(wBaseButton, value= 'Cancella', $
                uvalue='Canc', xsize= 150)

  WIDGET_CONTROL, wBase, /REALIZE
  WIDGET_CONTROL, wArea, GET_VALUE= wAreaID

  sInfo= { wAreaID: wAreaID $
         }

  WIDGET_CONTROL, wBase, SET_UVALUE= sInfo, /NO_COPY
  XMANAGER, 'Terzo programma', wBase, event= 'gestore_eventi'
end
```

Aspetti avanzati

Funzione CALL_EXTERNAL

La funzione *call_external* può essere utilizzata per invocare delle funzione scritte in altri linguaggi, come per esempio il linguaggio C. Vediamo un esempio di come si programma una *dll* in

ambiente windows (95/98/NT). Il seguente codice *C* è stato compilato con l'ambiente *MS Visual C++ 5.0*.

Esempio:

```
//
// ***   AF_DLLBEEP.C   ***
//
// 1. Costruire un nuovo progetto File/New…/Projects/Win 32 Dynamic−Link library
// 2. Includervi il seguente programma
// 3. Digitare
//   /EXPORT:MyBeep1,@1
//   /EXPORT:MyBeep2,@2
//   in Project/Setting.../Link/Project Options
// 4. Avviare il Bulder
 //
#include <windows.h>

// Function prototypes
BOOL WINAPI DllMain(HINSTANCE hInst, ULONG ulReason, LPVOID lpReserved);
LONG WINAPI MyBeep1(LONG lArgc, LPVOID lpvArgv);
        //Per suonare un beep di una certa frequenza e per una certa durata prefissate
LONG WINAPI MyBeep2(LONG lArgc, LPVOID lpvArgv);
        //Per suonare un beep la cui frequenza e durata sono passati come parametri

// Windows DLL entry point
BOOL WINAPI DllMain(HINSTANCE hInst, ULONG ulReason, LPVOID lpReserved)
{
        return(TRUE);
}

// Call to this dll function with no parameters
// lResult= CALL_EXTERNAL('af_dllbeep.dll', 'MyBeep1')
LONG WINAPI MyBeep1(LONG lArgc, LPVOID lpvArgv)
{
        Beep((DWORD)750, (DWORD)500);
        return(1);
}

// Call to this Win32 dll function passing:
// two LONG scaler by reference (2 parameters)
// theFreq= 300L & theTime= 200L
// lResult= CALL_EXTERNAL('af_dllbeep.dll', 'MyBeep2', theFreq, theTime)
LONG WINAPI MyBeep2(LONG lArgc, LPVOID lpvArgv)
{
        // lpvArgv e' un puntatore ad un array di 2 puntatori (in questo caso)
        DWORD *dwFreq= ((DWORD **)lpvArgv)[0];
        DWORD *dwTime= ((DWORD **)lpvArgv)[1];

        Beep(*dwFreq, *dwTime); //Hz, ms
        return(1);
}
```

Segue ora il programma scritto in IDL che richiama le funzione definite nella *dll*.

```
pro af_beep
  DLLpath= '\cpp\af_dllbeep\debug\'
  DLLname= 'af_dllbeep.dll'
  result= CALL_EXTERNAL(DLLpath+DLLname,'MyBeep1')
  freq= 300L & time= 300L
  result= CALL_EXTERNAL(DLLpath+DLLname,'MyBeep2', freq, time)
end
```

Funzione SPAWN

La funzione *spawn* può essere utilizzata per eseguire un comando del sistema operativo, un programma DOS o un'applicazione di tipo "*console*". A differenza del comando *$*, la funzione spawn può comparire anche all'interno di un programma IDL (mentre *$* si può solo immettere al prompt di IDL). IDL attende che il comando (o il programma) termini prima di proseguire con l'interpretazione e l'esecuzione delle istruzioni che seguono la chiamata alla funzione *spawn*.

Gli svantaggi della funzione *spawn* sono principalmente due:
1. la funzione apre una finestra di avvertimento dell'esecuzione del comando "esterno" che si chiude solo una volta che il programma termina
2. l'invocazione del comando induce l'apertura di una shell MS−DOS, in cui sono stampati gli eventuali output del comando.

Ciò determina una certa confusione sul desktop.

Esempio:

…
spawn("cd \idl50\mydir")
…

Risposte agli esercizi proposti

1.

```
IDL> varA = 0
IDL> help, varA
VARA          INT       =        0
IDL> varA = 10.0
IDL> help, varA
VARA          FLOAT     =      10.0000
IDL> varA = 'Hello world'
IDL> help, varA
VARA          STRING    = 'Hello world'
```

2.

```
<Expression>    COMPLEX   = (     5.50000,     3.50000)
```

3.

```
IDL> a= sindgen(5)
IDL> help, a
A             STRING    = Array[5]
IDL> print, a
       0       1       2       3       4
```

4.

```
IDL > read_jpeg, 'dna.jpg', img_dna
IDL > dim= size(img_dna)
IDL > dimx= dim[2]
```

```
IDL > dimy= dim[3]
IDL > tmp_plane= bytarr(dimx, dimy)
IDL > img_out= bytarr(3, dimx, dimy)
IDL > window, /free, xsize= dimx, ysize= dimy, title='Immagine originale'
IDL > TV, img_dna, true= 1
IDL > tmp_plane[*,*]= img_dna[0,*,*]
IDL > img_out[0,*,*]= median(tmp_plane, 3)
IDL > tmp_plane[*,*]= img_dna[1,*,*]
IDL > img_out[1,*,*]= median(tmp_plane, 3)
IDL > tmp_plane[*,*]= img_dna[2,*,*]
IDL > img_out[2,*,*]= median(tmp_plane, 3)
IDL > window, /free, xsize= dimx, ysize= dimy, title='Immagine filtrata'
IDL > TV, img_out, true= 1
```

5.

```
IDL > sobelx= ( [1., 0., -1.] # [1., 2., 1.] ) / 4.
IDL > sobely= ( [1., 2., 1.] # [1., 0., -1.] ) / 4.
```

6.

Sia *img* il nome dell'immagine a livelli di grigio.

```
hist= lonarr(256)
for x = 0 , (size(img))[1]-1 do begin
        for y = 0 , (size(img))[2]-1 do begin
                c= img[x, y]
                hist[c]= hist[c]+1
        endfor
endfor
```

7.

```
 for x= 0.0 , 1 , 0.1 do begin
        …
 endfor
```

8.

```
IDL > window, /free, xsize= (size(img_out))[1], ysize= (size(img_out))[2]
IDL > tvscl, img_out
```