

Premessa

“Raccolta Matlab #1” è un pdf contenente le versioni testuali e, ovviamente, il codice sorgente **25 videotutorial su Matlab** pubblicati gratuitamente da Francesco Milanese sul suo canale Youtube e sul sito www.francescomilanese.com.

L'ebook, che consta di 68 pagine in formato A4, ha quindi un prezzo volutamente basso perché si rivolge a chi ha seguito i videotutorial gratuiti, li ha trovati interessanti e desidera avere un testo in pdf (con screenshots dei videotutorial) da poter consultare facilmente, mediante le funzioni di ricerca o stampandolo... o desidera ringraziare in maniera concreta l'autore dei videotutorial, ottenendo comunque una piccola “guida – reference” sulle basi e alcune operazioni effettuabili in Matlab.

Nella prima parte sono presenti i 12 tutorial sulle basi di Matlab. Come detto nei video, questa serie di tutorial è rivolta a chi ha già qualche esperienza di programmazione (con altri linguaggi) e desidera avere una piccola “reference” su Matlab; non è un corso di programmazione da zero, ma una serie sulle basi di tale linguaggio e ambiente di sviluppo.

Nella seconda parte sono presenti 11 tutorial sull'elaborazione delle immagini digitali in Matlab. In questi tutorial non viene mostrata la teoria che sta dietro alle varie operazioni (trasformate di Fourier, filtri gaussiani, operatori morfologici, ...) ma solo le funzioni e gli script necessari per svolgere tali operazioni in Matlab.

Nella terza parte vi sono, infine, due tutorial “extra”: il primo è un articolo sulla risoluzione delle equazioni differenziali (ODE), pubblicato e disponibile in versione testuale sul sito web dell'autore; il secondo è un breve articolo sull'utilizzo di *i*, *j*, *li* e *lj* in Matlab.

Per il sommario completo: <http://www.francescomilanese.com/matlab-01-sommario.pdf>

SOMMARIO

Prima parte: le basi di Matlab

1. Introduzione. La GUI. L'help di sistema. Alcuni comandi di base	1
2. Variabili e operazioni matematiche di base. Variabili predefinite. Tutto è matrice	5
3. Operatori logici e operatori relazionali. Funzioni logiche. La notazione due punti	9
4. Matrici predefinite: ones, zeros, eye, rand	12
5. Funzioni matematiche predefinite	14
6. Controllo di flusso: i costrutti if e switch; i cicli for e while	16
7. Le funzioni in Matlab. File .m per funzioni o script	20
8. Grafici 2D con plot, subplot, semilogy; etichette per grafici ed assi; legende	23
9. Meshgrid. Grafici 3D con surf	26
10. Caricare un file (e un file immagine). Salvare un file (e un file immagine)	28
11. Operazioni di base sulle immagini: conversione tra formati	32
12. Operazioni di base sulle immagini: funzioni del gruppo im	34

Seconda parte: funzioni per l'elaborazione delle immagini digitali (E.I.D.)

1. Matlab per l'elaborazione delle immagini digitali: le basi	35
2. FFT: trasformata di Fourier (e IFFT, l'inversa)	38
3. Filtri gaussiani LP e HP	40
4. imfilter: media aritmetica e geometrica, mediano, midpoint, ...	42
5. imnoise e il rumore impulsivo	45
6. Il filtro di “degrado atmosferico” (Hufnagel – Stanley)	46
7. Il filtro di restauro Butterworth	48
8. Elementi strutturali e operatori morfologici	50
9. Immagine “sintetica” con figure da salvare su disco	52
10. Animazione casuale e FRAMES	54
11. Realizzare un filmato AVI da una sequenza di immagini	59

Extra

- | | |
|---|----|
| 1. Risoluzione delle equazioni differenziali con ODE (con esempio) | 62 |
| 2. Tips: perché i e j non dovrebbero essere usati come nomi di variabili? | 67 |

*	*	*
*	*	*

Le basi - Tutorial 1

Introduzione. La GUI. L'help di sistema. Alcuni comandi di base

Matlab (Matrix Laboratory) è un software che permette di definire ed eseguire **routines per il calcolo numerico**, ma non solo: è anche un vero e proprio linguaggio di compilazione ed interprete, tramite il quale è possibile **creare script e funzioni** dotate di controlli di flusso e in grado di leggere o salvare dati da/su file, incluse le immagini, i filmati e l'audio, tutti 'trattati' come sequenze (vettori o matrici) di valori numerici.

Una delle caratteristiche più interessanti (ed utili) di **Matlab** è la possibilità di estenderlo con delle **toolbox**, che sono **raccolte di funzioni** ("packages" o "librerie", in altri linguaggi) aggiuntive (il 'core' di **Matlab** è, comunque, già abbastanza ricco e mette a disposizione molte funzioni di base); ne esistono centinaia e consentono di utilizzare funzioni complesse o di interagire con vari tipi di dati (es.: filmati, clip audio, ...).

La **GUI** (interfaccia) di **Matlab** di base, una volta avviato il programma, mostra - di default - all'interno della finestra principale le seguenti schede:

- **la barra degli strumenti**, contenente - tra le altre cose - il percorso della **Directory di lavoro di Matlab**: i file da caricare (file di script o funzioni .m, o file delle risorse quali immagini e clip audio) dovranno trovarsi in questa cartella per essere caricati ed utilizzati, e i file generati mediante **Matlab** verranno salvati qui;

- **la finestra del Workspace**, lett.: spazio di lavoro, che elenca le **variabili attualmente utilizzate**, la loro dimensione (sia dal punto di vista "della matrice", come vedremo a breve, che in termini di memoria) e il loro tipo (valori interi, double, frames, ...);
- **la Command Window**, il prompt dei comandi di **Matlab**: è qui che, scrivendo un'istruzione a destra del prompt (il “terminale”) `>>` e premendo Invio, potrete **eseguire dei comandi in Matlab**;
- **la Command History**, che riepiloga le ultime istruzioni inserite nella Command Window, senza però mostrare l'eventuale output a video visibile nella Command Window.

Altre schede/pannelli possono essere attivati in seguito, ad esempio per **visualizzare un grafico (plot)** o una immagine caricata da file o generata mediante lo stesso **Matlab** a partire da una matrice di valori.

Matlab dispone di un ricco **help online**, richiamabile scrivendo

```
>> help [nomecomando]
```

nella **Command Window** e premendo Invio.

Scrivendo semplicemente

```
>> help
```

e premendo Invio, **Matlab** restituirà l'elenco dei "topic" (argomenti, capitoli) dell'help. A seconda del numero e del tipo di toolbox presenti, l'elenco restituito potrà essere più o meno lungo.

Se, invece, non conoscete un particolare comando ma **sapete "di cosa tratta" e volete rintracciarlo**, potete utilizzare l'istruzione:

```
>> lookfor [parola chiave];
```

ad esempio, per cercare il comando del valore assoluto (, *absolute value* in inglese), digitando:

```
>> lookfor absolute
```

verranno mostrati, a video, i comandi nella cui descrizione sarà contenuta la parola **absolute**; successivamente, potremo chiamare l'help su una di queste istruzioni per saperne di più (comunque, la funzione nativa di Matlab da utilizzare per calcolare il valore assoluto di un numero n è **ABS(n)**).

Ovviamente, online sono reperibili (specialmente per le varie toolbox aggiuntive) **molti manuali**.

Per **liberare il Workspace** (cancellando tutte le variabili e gli "oggetti creati" e i relativi valori), digitare nella **Command View**:

```
>> clear
```

e premere Invio; tale comando può essere utilizzato anche in maniera "mirata" per eliminare, dal **Workspace**, una determinata variabile: digitare ad esempio

```
>> clear x
```

nella **Command View** e premere Invio per eliminare dal **Workspace** la variabile x .

Per **pulire la Command Window**, invece, digitare nella stessa:

```
>> clc
```

e premere Invio.

Alcuni comandi permettono di **interagire con il file system**, il cui punto d'accesso per **Matlab** è rappresentato dalla **Current Directory**; digitate, ad esempio:

```
>> dir
```

nella **Command Window** e premete Invio per ottenere l'elenco dei file presenti nella **Current Directory**.

Da notare che l'**help** di **Matlab** vi suggerisce anche comandi "simili" (per operazione effettuata o per "contesto") a quello passato come parametro; digitate ad esempio

```
>> help dir
```

nella **Command Window** e premete Invio, ed individuate la sezione **"See Also"** nell'output: le istruzioni seguenti sono istruzioni **Matlab** legate in qualche modo a **dir**.

Nella **Command Window** possiamo, quindi, inserire le nostre istruzioni; per 'spezzare' un'istruzione su più righe (ad esempio, se è molto lunga), utilizzare:

...

Concludendo un'istruzione con il punto e virgola ; , l'output dell'operazione effettuata NON verrà mostrato a video; ad esempio, scrivendo:

```
x=5;
```

e premendo Invio non verrà mostrato nulla, ma scrivendo:

```
x=5
```

e premendo Invio, la **Command Window** mostrerà:

```
x =
```

```
5
```

ossia ci mostrerà l'ultima operazione effettuata: alla variabile *x* è stato assegnato il valore 5.

Con la **virgola** separeremo, a seconda dei casi, dei **valori** (ad esempio, per definire gli elementi di un vettore, come vedremo a breve) o delle **istruzioni** (è possibile, quindi, scrivere più istruzioni in un'unica riga), mentre con **%** attiveremo la modalità commento sulla riga corrente: tutti i caratteri e le istruzioni di una riga posti dopo **%** verranno **ignorati dall'interprete Matlab**.

Tutti questi comandi (e molti altri) verranno comunque (ri)discussi nel corso della guida, per cui niente paura se ora il tutto può apparire confusionario.

* * *

Le basi - Tutorial 2

Variabili e operazioni matematiche di base. Variabili predefinite. Tutto è matrice

In **Matlab**, le **variabili** vengono create semplicemente assegnando ad una sequenza di caratteri (il nome della variabile) un valore; ad esempio, digitando nel prompt:

```
>> a = 5;
```

creeremo una variabile di nome *a* con valore 5.

Nella finestra del **Workspace** vedremo apparire la variabile *a*, descritta come un **array (vettore) di double** (**Matlab** ha definito per noi il tipo di variabile; comunque, vedremo in seguito come effettuare delle conversioni)... come, un **array**? Sì, perchè in **Matlab** **"tutto è una matrice"**: le operazioni sono da intendersi come operazioni tra matrici e, quindi, una singola variabile diventa una matrice di dimensione 1x1, e va memorizzata mediante **un array (di dimensione 1)**.

I numeri puri (valori **scalari**), quindi, saranno **matrici 1x1**; i vettori, poi, saranno particolari matrici 1xN (vettore riga) o Mx1 (vettore colonna), mentre avremo infine le matrici vere e proprie, MxN (M righe per N colonne).

Per **creare un vettore riga**, bisogna inserire i valori all'interno delle parentesi quadre [] separandoli mediante una virgola (,) ; ad esempio:

```
>> vettoreRiga = [3, 2, 5];
```

La virgola indica un nuovo valore sulla stessa riga, mentre **il punto e virgola indica il passaggio ad una riga successiva**, per cui creeremo un vettore colonna con (esempio):

```
>> vettoreColonna = [3; 2; 5];
```

e una **matrice** vera e propria con (esempio):

```
>> miaMatrice = [3, 2, 1; 4, 5, 6; 9, 10, 12];
```

I nomi delle variabili sono **case sensitive**, per cui la variabile

```
>> A = a * 5;
```

sarà diversa da *a*, che continuerà ad esistere, con il proprio valore.

Il punto e virgola finale, nelle nostre istruzioni, evita di mostrare sul prompt l'esito dell'operazione (es: il valore assegnato ad una variabile), ma provate a scrivere ad esempio:

```
>> a = 5;
```

```
>> a
```

(premendo Invio dopo aver premuto *a*) per vedere mostrato a video il valore della variabile (potete farlo, ad esempio, anche con "vettoreColonna" per vedere, stampati in verticale - in colonna, appunto - i valori della variabile).

I nomi delle variabili possono avere al massimo 63 caratteri e devono iniziare con una lettera, che può essere però seguita da lettere, numeri o dal simbolo `_` (underscore, il trattino basso).

Alcune variabili sono predefinite, per cui i loro nomi sono "riservati"; ad esempio, provate a digitare i valori elencati nella tabella seguente nel prompt, senza ; finale (in modo da vedere l'output "ans" restituito da Matlab) e a premere Invio:

ans	Il nome della variabile che contiene il risultato dell'ultima operazione effettuata (se non assegnata ad una variabile)
pi	Pi greco

inf	Il valore infinito
NaN	"Non un numero", cioè indefinito o indeterminato, come ad esempio il risultato di 0/0
i	Radice quadrata di -1 (o: 0 + 1i)
j	Radice quadrata di -1 (o: 0 + 1i)
eps	Precisione di macchina
realmin	Il più piccolo valore intero positivo utilizzabile / rappresentabile
realmax	Il più grande valore intero positivo utilizzabile / rappresentabile

Le **operazioni matematiche di base** sono invece le seguenti:

+	Addizione; a+b
-	Sottrazione; a-b
*	Prodotto; a*b
.*	Prodotto termine a termine per i vettori; a.*b
/	Divisione; a/b
./	Divisione termine a termine per i vettori; a./b
^	Elevamento a potenza; a^b
.^	Elevamento a potenza termine a termine per i vettori; a.^b

Un esempio per chiarire meglio le differenze tra le operazioni con e senza **.** (per scalari o per vettori): data una **matrice QUADRATA** di nome **X**, con:

```
>> Y = X^2;
```

avremo, in **Y**, il prodotto della matrice **X** con sè stessa, mentre con:

```
>> Z = X.^2;
```

avremo, in **Z**, termini di **X** elevati "singolarmente" al quadrato ($Z_{i,j} = (X_{i,j})^2$).

Un'ultima considerazione sui tipi di dati: ecco i tipi primitivi di **Matlab**:

- **logical**; un valore booleano (0 o 1), memorizzato con 1 byte per elemento.
- **char**; elemento "letterale" (es: 'a'), memorizzato con 2 byte.
- **uint8**; valore intero senza segno, memorizzato con 1 byte (range [0, 255]).

- **uint16**; valore intero senza segno, memorizzato con 2 byte (range [0, 65535]).
- **uint32**; valore intero senza segno, memorizzato con 4 byte (range [0, 4294967295]).
- **int8**; valore intero con segno, memorizzato con 1 byte (range [-128, 127]).
- **int16**; valore intero con segno, memorizzato con 2 byte (range [-32768, 32767]).
- **int32**; valore intero con segno, memorizzato con 4 byte (range [-2147483648, 2147483647]).
- **single**; valore float, memorizzato con 4 byte (range $[-10^{38}, 10^{38}]$).
- **double**; valore double memorizzato con 8 byte per elemento (range $[-10^{308}, 10^{308}]$).

Esistono anche **funzioni** che consentono (qualora sia possibile effettuare tale operazione) di **convertire (to cast) un dato da un tipo ad un altro**: per effettuare il **casting**, in genere basta passare un valore come parametro ad una funzione che ha, per nome, il nome del tipo di dati, ad esempio con:

```
>> x = 3;  
  
>> a = uint8(x);  
  
>> c = double(a);
```

Esistono anche funzioni per convertire tipi di dati più complessi tra loro o per "ridurli" a tipi di dati primitivi (matrici): ne ripareremo in seguito.

Discorso a parte merita la funzione **real(numeroComplesso)**, che permette di estrapolare solo la parte intera di un numero complesso passato come parametro; in maniera analoga, la funzione **imag(numeroComplesso)** permette di estrapolare e depositare come valore a sè stante (matrice 1x1) la parte immaginaria di un numero complesso, mentre **conj(numeroComplesso)** calcola e restituisce il complesso coniugato di un numero complesso.

* * *

Le basi - Tutorial 3

Operatori logici e operatori relazionali. Funzioni logiche. La notazione due punti

Gli **operatori logici** supportati da **Matlab** sono 3:

~	Il NOT, la negazione logica
&	L'AND logico
	L'OR logico

Il **TRUE** è indicato con 1, **FALSE** con 0.

Matlab supporta poi 6 **operatori relazionali**:

<	Minore di
<=	Minore di o uguale a
>	Maggiore di.
>=	Maggiori di o uguale a
==	Uguale a
~=	Diverso da

Ad esempio, l'espressione:

```
>> 3 < 5;
```

depositerà, nella variabile **ans**, il valore 1 (true), così come, ad esempio:

```
>> 3 ~= 5;
```

depositerà, nella variabile **ans**, il valore 0 (false).

Abbiamo, poi, delle vere e proprie **FUNZIONI LOGICHE**, che prendono in input delle **ESPRESSIONI LOGICHE**:

xor(a, b)	Effettua lo XOR (or esclusivo) di a e b
any(x)	Restituisce 1 (true) se c'è qualche elemento di x diverso da zero
all(x)	Restituisce 1 (true) se tutti gli elementi di x sono diversi da zero
isnan(x)	Restituisce 1 (true) per ciascun valore NaN in x
isinf(x)	Restituisce 1 (true) per ciascun valore INF (infinito) in x
finite(x)	Restituisce 1 (true) per ciascun valore finito in x

Come detto precedentemente, l'output varrà 1 per **TRUE**, 0 per **FALSE**.

Parliamo, ora, di un argomento molto importante in **Matlab**: la **"notazione due punti"**.

Il carattere **:** serve ad indicare la presenza di un **CICLO IMPLICITO**, utilissimo per **creare o scorrere dei vettori** (e, se usato in cicli annidati, matrici a più dimensioni); ad esempio, con:

```
>> x = 1:5;
```

creiamo un ciclo che va a "passi" di 1 (valore di default) da 1 a 5; i valori generati vengono assegnati a **x**, che quindi è un **vettore riga**, e difatti digitando **x** sul **prompt** avremo quest'output:

```
1 2 3 4 5
```

Il "passo" di default è, quindi, 1, ma possiamo specificare un valore differente, da porre in mezzo agli estremi del **ciclo**; ad esempio, con:

```
>> y = 1 : .2 : 2;
```

creeremo un **vettore riga** con valori da 1 a 2 a passi di 0,2 , per cui digitando **y** sul **prompt** e premendo Invio vedremo a video:

```
1.0000 1.2000 1.4000 1.6000 1.8000 2.0000
```

Possiamo poi utilizzare la notazione `:` anche per assegnare "in un colpo" dei valori a parti di un vettore; ad esempio, digitando:

```
>> y(1:3) = 0;
```

assegneremo il valore 0 ai primi tre elementi di **y**, per cui digitando sul **prompt y** e premendo Invio vedremo, a video, i seguenti valori:

```
0 0 0 1.6000 1.8000 2.0000
```

Possiamo anche **modificare "in un colpo solo" tutti i valori di un vettore** grazie al ciclo implicito di `:`; ad esempio, scrivendo:

```
>> x(:) = 1;
```

porremo a 1 **tutti i valori del vettore x**.

* * *

Le basi - Tutorial 4

Matrici predefinite: ones, zeros, eye, rand

Matlab permette di definire velocemente delle **matrici**, ma mette a disposizione delle **funzioni per generare delle matrici** "tipiche" in maniera nativa, evitando al programmatore di doverle implementare; in particolare, abbiamo le seguenti funzioni:

EYE(N); EYE(M, N) ; EYE(SIZE(A))	Genera una matrice identità (diagonale a 1, tutti gli altri valori a zero), nel primo caso di dimensione NxN, nel secondo MxN e nel terzo con dimensioni uguali a quelle della matrice A
ZEROS(N); ZEROS(M,N); ZEROS(SIZE(A))	Genera una matrice di zeri, nel primo caso di dimensione NxN, nel secondo MxN e nel terzo con dimensioni uguali a quelle della matrice A
ONES(N); ONES(M,N); ONES(SIZE(A))	Genera una matrice di elementi, tutti con valore 1, nel primo caso di dimensione NxN, nel secondo MxN e nel terzo con dimensioni uguali a quelle della matrice A
RAND(N); RAND(M,N); RAND(SIZE(A))	Genera una matrice di elementi dal valore casuale, nel primo caso di dimensione NxN, nel secondo MxN e nel terzo con dimensioni uguali a quelle della matrice A

Da notare che possiamo "depositare" una matrice così generata anche in **ans**, digitando ad esempio:

```
>> eye(3);
```

Per generare **matrici con valori casuali (random)** sono disponibili altre **funzioni**, oltre a **rand**; si consulti, in tal senso, l'**help di Matlab**, digitando ad esempio

```
>> help rand
```

* * *

Le basi - Tutorial 5

Funzioni matematiche predefinite

Oltre alle matrici predefinite, Matlab mette a disposizione un **set di funzioni matematiche di base**, quali ad esempio (nota: le seguenti funzioni sono tutte **unarie**):

sin	Restituisce il seno del valore passato come parametro
cos	Restituisce il coseno del valore passato come parametro
asin	Restituisce l'arcoseno del valore passato come parametro
acos	Restituisce l'arcocoseno del valore passato come parametro
tan	Restituisce la tangente del valore passato come parametro
atan	Restituisce l'arcotangente del valore passato come parametro
exp	Funzione esponenziale (e^n , n è il parametro passato)
log	Calcola il logaritmo naturale del valore passato come parametro
sqrt	Restituisce la radice quadrata del valore passato come parametro
abs	Restituisce il valore assoluto del numero passato come parametro
sign	Implementa la funzione segno
log10	Calcola il logaritmo in base 10 del valore passato come parametro
log2	Calcola il logaritmo in base 2 del valore passato come parametro

Altre **funzioni** interessanti:

- **linspace(inizio, fine, numero di punti)**

Consente di creare un vettore con (fine-inizio) elementi equispaziati; ad esempio, scrivendo:

```
>> x = linspace(0, 1, 5)
```

otterremo il **vettore riga** x così composto:

```
0 0.2500 0.5000 0.7500 1.0000
```

Come avrete notato, il risultato di molte operazioni mostra valori con 4 **cifre decimali**; tramite il comando ***format***, è possibile cambiare questa impostazione (che è quella di default), digitando ad esempio:

```
>> format long
```

Per maggiori informazioni sui possibili parametri di ***format***, si rimanda all'**help di Matlab** (sarà sufficiente digitare *help format* e premere Invio nella **Command Window**).

Infine, la funzione ***length***, che - dato un **vettore**, ad esempio x - restituisce il **numero di elementi contenuti nel vettore**:

```
>> length(x)
```

E per le **matrici** a due o più dimensioni?

In tal caso, bisognerà utilizzare ***size***:

```
>> size(A);
```

che restituirà un **array di valori**: ad ogni cella corrisponderà il numero di elementi della relativa dimensione (es.: 1 3 indicherà una matrice di 1 riga con 3 colonne).

* * *

Le basi - Tutorial 6

Controllo di flusso: i costrutti if e switch; i cicli for e while

Per **controllare il flusso dell'esecuzione** o effettuare controlli su condizioni, **Matlab** mette a disposizione, come molti altri linguaggi, i costrutti:

- **if-else-end;**
- **switch;**
- **for;**
- **while.**

Iniziamo dall'**if**, che verifica se una condizione passata come parametro è "vera", ed in tal caso effettua un'azione; in caso contrario ("**else**") ne effettua un'altra (opzionale).

È possibile avvalersi, inoltre, di **elseif**, che implementa un nuovo blocco **if** all'interno del precedente, per cui è possibile costruire **if annidati**.

La **sintassi** è la seguente:

```
IF expression
statements
ELSEIF expression
statements
ELSE
statements
END
```

Esempio pratico:

```
if 2 == 3
x=4;
elseif 2 < 3
x=5;
else
x=6;
end
```

Digitando **x** e premendo Invio, vedremo a video, come valore di **x**:

5

in quanto la condizione vera (**true**) è quella **dell'elseif**: $2 < 3$.

Il costrutto **switch** consente anch'esso, come **if**, di valutare un'espressione O UN VALORE passati come parametro e di intraprendere, in base al valore di tale espressione o del parametro, azioni differenti; a differenza dell'**if**, comunque, è più "ordinato": le varie possibilità vengono individuate da righe che iniziano con la parola case e le azioni da intraprendere vengono scritte di seguito.

Sintassi di Switch:

```
SWITCH switch_expr
CASE case_expr,
statement, ..., statement
CASE {case_expr1, case_expr2, case_expr3,...}
statement, ..., statement
...
OTHERWISE,
statement, ..., statement
END
```

Ad esempio, digitando:

```
switch (2+3)
case {2,3,4}
x = -1;
case 5
x = 0;
otherwise
x=1;
end
```

Il valore depositato, alla fine, in **x**, sarà 0.

Il **ciclo for** ripete una o più azioni *n* volte, dove **n** è dato dall'indice di terminazione meno l'indice di partenza del ciclo diviso il "passo" (che, di default, vale 1).

È possibile **annidare i cicli for**, ad esempio per **scorrere una matrice** a due o più dimensioni.

Sintassi del ciclo *for* in Matlab:

```
FOR variable = expr, statement, ..., statement END
```

Esempio pratico: **due cicli for, annidati**, che pongono a 1 tutti i valori di una matrice “**A**” 4x4:

```
n = 4;
for i=1:n
for j=1:n
A(i,j) = 1;
end
end
```

Provate, ad esempio, a modificare il ciclo **for** appena esposto per generare una **matrice triangolare inferiore** o superiore.

Il ciclo **while** è simile al ciclo **for**, ma non dispone di una informazione sull'incremento nella sua dichiarazione, per cui toccherà a voi gestire questo aspetto (l'esecuzione del ciclo potrebbe non avere mai fine!).

La **sintassi** è la seguente:

```
WHILE expression
statements
END
```

Il seguente esempio **incrementa un contatore** fino quando esso non vale 10 (e, arrivati a quella condizione, il ciclo **while** si interrompe, per cui il flusso dell'esecuzione ritorna al resto dello script):

```
i = 0;
while (i<10)
i = i+1;
end
```

* * *

Le basi - Tutorial 7

Le funzioni in Matlab. File .m per funzioni o script

Matlab ci consente di **definire funzioni e script** e di salvare questi elementi in appositi file, di estensione **.m**, riconoscibili dal programma come file contenenti, appunto, istruzioni da eseguire.

È quindi possibile **definire**:

- **script**, ossia **file di comandi**, che non hanno parametri di input/output, ma solo un insieme di **operazioni da effettuare sul workspace**;
- **funzioni**, che accettano parametri in input, li elaborano e restituiscono un risultato.

In entrambi i casi, i **file .m** saranno "in chiaro", per cui potrete scrivere i vostri file **.m** con un qualsiasi **editor di testo**.

All'interno di questi file è possibile (anzi, è raccomandabile) **inserire dei commenti**; le righe dei commenti dovranno iniziare tutte con il carattere **%**.

Matlab ignora i commenti, che servono solo per lo sviluppatore, ad esempio per spiegare come funziona una... funzione o che tipo di output restituisce.

Le **funzioni** hanno la seguente **struttura**:

- iniziano con una riga con la parola chiave ***function***, il parametro da restituire in **output**, il carattere = (istruzione di assegnamento), il **nome della funzione** e l'eventuale lista di **parametri** tra () e separati da virgole;
- il **corpo** della funzione.

Sarà possibile inserire, poi, righe di commento prima, dentro e dopo le istruzioni della funzione, nel file.

Segue un esempio di file di script che non fa altro che creare una nuova variabile di nome **a** (attenzione: sovrascriverà il valore di una *eventuale* variabile di nome **a**, se presente nel workspace!), stamperà nella **Command Window** una **stringa** (si noti l'utilizzo della funzione `display`) e, in seguito, il valore di **a**:

```
% File di script di prova
a = 5;
display('Ciao mondo !');
a
```

Potete copiare queste righe così come sono in un file di testo vuoto, salvarlo con **estensione .m** nella directory di lavoro di **Matlab** (o da qualche altra parte, ma in tal caso dovrete cambiare la directory di lavoro di Matlab per "trovarlo" ed eseguirlo) e digitare semplicemente, nella **Command Window**, il nome del file SENZA ESTENSIONE, per eseguirlo.

Ecco invece un **esempio di file funzione** che, presa in input una **matrice A**, calcola la radice quadrata dei suoi elementi e deposita il risultato nella matrice **B**:

```
function B=radice(A)
B = sqrt(A);
```

Per poter utilizzare questa funzione, il file che la conterrà **dovrà trovarsi nella directory di lavoro di Matlab**; a quel punto, vi basterà scrivere il nome della funzione con i suoi parametri, scrivendo

ad esempio:

```
A = rand(4,3);  
B = radice(A);
```

per eseguirla e vedere il risultato a video.

* * *

Le basi - Tutorial 8

Grafici 2D con plot, subplot, semilogy; etichette per grafici ed assi; legende

Con **Matlab** è possibile generare velocemente dei **grafici** visualizzabili a video.

Per **generare un grafico**, è necessario predisporre un **vettore di valori** per l'asse **X** e un **vettore di valori** per l'asse **Y**, dopodiché bisognerà utilizzare l'istruzione **plot**:

```
plot(vettoreValoriX, vettoreValoriY, [eventuale definizione del tratto  
e del colore]).
```

Il terzo parametro (opzionale) di questa istruzione permette di definire **come disegnare la curva nel grafico**, impostando il colore della curva e "cosa" disegnare; l'elenco dei possibili valori è abbastanza lungo, per cui si rimanda all'**help online** del comando **plot**.

Per fornire un'**etichetta testuale** all'asse delle **X**, all'asse delle **Y** o al grafico stesso, utilizzare:

```
>> xlabel ('Etichetta asse x');  
>> ylabel ('Etichetta asse y');  
>> title ('Titolo del grafico');
```

Esempio pratico (potete ricopiarlo così com'è nella vostra **Command Window** e premere Invio per eseguirlo):

```
>> t = 0 : 0.01 : 2*pi;  
>> x = cos(t);
```

```
>> y = sin(t);  
>> plot(x,y, 'b*');
```

Eseguito così, **plot** mostrerà **una sola curva per volta**; per **visualizzare più curve nello stesso grafico**, utilizzare il comando **hold on**.

Esempio di più curve in un plot (ricopiate nella vostra **Command Window** e premete Invio):

```
>> t = 0 : 0.01 : 2*pi;  
>> x = cos(t);  
>> y = sin(t);  
>> plot(x,y, 'b*');  
>> hold on;  
>> x2 = 1:5;  
>> y2 = x2*2;  
>> plot(x2, y2, 'g+');
```

In genere, **Matlab** **scalerà gli assi** - o ne visualizzerà solo una porzione - per rendere il grafico ben visibile; **per mantenere gli assi uguali**, digitare **axis equal** dopo l'istruzione **plot**:

```
>> axis equal
```

Per utilizzare una **scala logaritmica** (base 10) per l'asse delle **Y** utilizzare, al posto di **plot**, l'istruzione **semilogy**:

```
>> semilogy(vettoreValoriX, vettoreValoriY, [eventuale definizione del  
tratto e del colore]).
```

per cui il nostro **script** di prima diventerà, ad esempio:

```
>> t = 0 : 0.01 : 2*pi;  
>> x = cos(t);  
>> y = sin(t);  
>> semilogy(x,y, 'b*');  
>> hold on;  
>> x2 = 1:5;  
>> y2 = x2*2;  
>> semilogy(x2, y2, 'g+');
```

È possibile **visualizzare più finestre** di *plot* o *semilogy* all'interno di un'unica finestra di output **Matlab**, mediante *subplot*.

L'istruzione:

```
subplot(M, N, K);
```

crea una figura contenente **M*N grafici** (M righe per N colonne), mentre K indica che le istruzioni CHE SEGUONO si riferiscono al **K-esimo grafico** tra quelli da mostrare a video (nel numerare i grafici di una finestra *subplot*, si parte da 1 considerando come "primo" il grafico in alto a sinistra e si scorrono le righe da sinistra a destra, riga per riga); esempio pratico (copiatelo nella vostra **Command Window** e premete Invio):

```
>> t = 0 : 0.01 : 2*pi;  
>> x1 = cos(t);  
>> y1 = sin(t);  
>> x2 = 1:5;  
>> y2 = x2*2;  
>> x3 = 2:10;  
>> y3 = x3/3;  
>> x4 = 3:8;  
>> y4 = x4*3/2;  
>> subplot(2, 2, 1);  
>> plot(x1, y1, 'gP');  
>> subplot(2, 2, 2);  
>> plot(x2, y2, 'b*');  
>> subplot(2, 2, 3);  
>> plot(x3, y3, 'r-');  
>> subplot(2, 2, 4);  
>> plot(x4, y4);
```

Da notare che le finestre di output mostrate da **Matlab** per *plot*, *semilogy* e *subplot* mettono a disposizione il tasto '*salva*', per salvare i grafici generati sotto forma di immagini in formato **.fig**, e il tasto '*stampa*', appunto per stampare i grafici.

* * *

Le basi - Tutorial 9

Meshgrid. Grafici 3D con surf

Matlab ci consente anche di **creare grafici tridimensionali** (e, in questo caso, parleremo di *superfici*) a partire da matrici di valori.

Per generare una **superficie**, avremo bisogno di **una griglia di valori** creata a partire dai vettori ***X*** e ***Y***; per generare una tale griglia, utilizziamo la funzione ***meshgrid*** di **Matlab**:

$$[X, Y] = \text{MESHGRID}(x, y)$$

che **trasforma i valori dei vettori *x* e *y***, passati come parametri, in **array *X* e *Y***, utilizzabili per valutare funzioni o, come nel nostro caso, **per generare plot 3D**.

A partire da ***X*** e ***Y*** potremo, quindi, calcolare i valori di ***Z***, che definirà "l'altezza" di un "vertice" nel nostro **plot 3D**.

Per valutare - e rappresentare a video mediante **grafico 3D** - la **funzione**:

$$z = x(1-x)y(1-y)$$

scriveremo quindi (copiate questa porzione di codice nella vostra **Command Window** e premete Invio):

```
>> n=5;  
>> m=5;  
>> x=linspace(0,1,n);  
>> y=linspace(0,1,m);  
>> [X,Y] = meshgrid(x,y);  
>> Z = X.*(1-X).*Y.*(1-Y);  
>> surf(X,Y,Z);
```

La funzione che ci consente di **generare il grafico 3D** e di mostrarlo a video è, quindi, **surf**.

Esistono comunque anche altre **funzioni per le superfici**, come ad esempio:

view	Consente di modificare l'orientamento del grafico
colormap	Consente di definire il colore del grafico
shading	Permette di cambiare le impostazioni di ombreggiatura (per modificare la resa visiva) del grafico
mesh	Permette di disegnare un grafico "a griglia"
contour	Permette di disegnare un grafico del tipo "a curve di livello" (2D, visto dall'alto)
surf	Disegna la superficie 3D (è quello visto nell'esempio)

Per tutte queste funzioni esistono anche dei parametri utilizzabili per modificare l'aspetto finale dei grafici; per una trattazione più approfondita, si rimanda all'**help online di Matlab** disponibile per i vari comandi (es.: **help surf**, ...).

* * *

Le basi - Tutorial 10

Caricare un file (e un file immagine). Salvare un file (e un file immagine)

Matlab ci consente di **caricare interi file nel Workspace**, trattandoli mediante un'unica **variabile** (un *handler*, un gestore) che rappresenterà i dati sotto forma di matrici.

Le modifiche effettuate sull'handler non influenzeranno direttamente il file sorgente, in quanto **l'handler sarà una copia** in memoria dello stesso; dovremo, quindi, **salvare le modifiche su file** per avere una copia permanente (su disco) del nostro lavoro.

Il file da caricare dovrà trovarsi nella **Directory di lavoro di Matlab**, ed è qui che verranno salvati (se inseriremo comandi di salvataggio su file, ovviamente).

Per caricare un file, Matlab mette a disposizione i metodi:

- `load(' [nomeFile.estensione] ');`
alquanto "generico" e con delle restrizioni (ma anche parametri per impostarne il comportamento; si consulti, a tal proposito, l'help online);
- `imread(' [nomeFileImmagine.estensione] ');`
per **caricare file di immagini**.

Una volta caricato un file immagine, ad esempio con **imread** (per eseguire correttamente lo script che segue, mettete un file **'prova.jpg'** nella vostra **directory di lavoro** o cambiate il nome della variabile nello script che segue):

```
>> immagine = imread('prova.jpg');
```

Matlab tratterà il dato come una **matrice di valori**, per cui scrivendo:

```
>> length(immagine);
```

avremo il numero di colonne (la lunghezza di un vettore riga dell'immagine; corrisponde alla **larghezza, in pixel, dell'immagine**), mentre scrivendo:

```
>> size(immagine);
```

avremo **tre valori**: numero di righe (**altezza** dell'immagine), numero di colonne (**larghezza** dell'immagine), **numero di livelli** (ad esempio, per immagini a colori **RGB**, **3**: uno per canale-colore).

*[NOTA --- In **Matlab** il 'sistema di riferimento' delle immagini come matrici parte dall'angolo in alto a sinistra, per cui la cella di indice 0,0 sarà quella in alto a sinistra nell'immagine.]*

Per conoscere, ad esempio, il **valore di verde** (secondo canale; in **Matlab** il primo indice di un array è 1) del pixel posto nella quarta colonna della terza riga, scrivere:

```
>> valorePixel = immagine(3, 4, 2)
```

Il valore ottenuto dovrà essere trattato in base al tipo di dato che rappresenta l'immagine; se, ad esempio, si tratterà di un **uint8**, dovremo aspettarci valori nel **range [0, 255]**.

Matlab supporta diversi **formati di file immagine** (altri possono essere trattati mediante **toolbox** opportune); tra questi, abbiamo ad esempio:

TIFF (.tif, .tiff); **JPEG** (.jpg, .jpeg); **GIF** (.gif); **BMP** (.bmp); **PNG** (.png); **XWD** (.xwd).

Dopo aver lavorato su un'immagine, ad esempio applicandole un **filtro** (vd. tutorial della seconda parte), la cosa più logica da fare è **salvare il tutto su file**, cosa che potete fare con ***imwrite***:

```
>> imwrite(matriceImmagine, nomeFile, estensioneFile);
```

che salva, appunto, la **MATRICE** depositata nella variabile **handler**.

È possibile utilizzare una gran quantità di valori per i **parametri di *imwrite***, per cui si rimanda all'**help online** di tale comando per una trattazione più approfondita.

Abbiamo detto che ***imwrite*** salva la matrice "convertendola" in immagine, per cui potremmo pensare di generare matrici mediante procedure o formule e salvarle come immagini su file...

... supposizione esatta! La seguente porzione di codice (ottimizzabile, come esercizio!), ad esempio, **genera una "scacchiera"** (8x8 caselle, ciascuna di 16x16 pixel) e **la salva nella current directory con nome "scacchiera.bmp"**:

```
>> % Alloco una matrice di zeri di dimensione 128x128
>> scacchiera = zeros(128,128);

>> % Una casella nera è una matrice di zeros 16x16
>> cN = zeros(16,16);

>> % Una casella bianca è una matrice di ones 16x16
>> cB = ones(16,16);

>> % La prima riga sarà:
>> primaRiga = [cB, cN, cB, cN, cB, cN, cB, cN];
>> % La seconda riga sarà:
>> secondaRiga = [cN, cB, cN, cB, cN, cB, cN, cB];
>> % La scacchiera sarà:
>> scacchiera = [primaRiga; secondaRiga; primaRiga; secondaRiga;
primaRiga; secondaRiga; primaRiga; secondaRiga];

>> % Salvo su file tale matrice. Diverrà un'immagine.
>> imwrite(scacchiera, 'scacchiera.bmp', 'bmp');
```

Prima di salvare un'immagine, comunque, potreste volerla **vedere a video in Matlab**; utilizzando il comando ***imshow*** (utilizzabile anche in combinazione con ***subplot***, per vedere nella stessa finestra immagini e/o grafici):

```
>> imshow(nomeMatrice);
```

potrete vedere, nella finestra che apparirà a video, l'immagine rappresentata dalla matrice sulla quale state operando (provate, riprendendo lo script appena visto, ad eseguire ***imshow(scacchiera)***).

Così come è possibile salvare immagini scrivendole su file, è possibile scrivere altri tipi di dati in maniera permanente.

La funzione ***SAVE*** serve a salvare *TUTTO IL WORKSPACE* in un file, digitando:

```
>> save [nomeFile];
```

Per limitare il **salvataggio ad una o più variabili** (es: salvare solo le variabili **X** e **VALORE2**), scrivere ad esempio:

```
>> save [nomeFile] [X] [VALORE2];
```

Esistono, ovviamente, molti **parametri per *save*** e, comunque, molte altre **funzioni per gestire l'I/O da/su file**; per una trattazione più approfondita, si rimanda all'**help online di Matlab**.

* * *

Le basi - Tutorial 11

Operazioni di base sulle immagini: conversioni tra formati

Matlab mette a disposizione diverse **funzioni per effettuare conversioni tra formati**, utilissime per trattare le **immagini** (e non solo); abbiamo, ad esempio, le seguenti:

im2uint8	Conversione da immagine a matrice di uint8
im2uint16	Conversione da immagine a matrice di uint16
mat2gray	Conversione da immagine a matrice di double con valori nel range [0,1]
im2double	Conversione da immagine a matrice di double
im2bw	Conversione da immagine a matrice con tipo di dati "logical" (solo valori 0 e 1)
rgb2gray	Conversione dal modello di colore rgb a scala di grigi
rgb2ycbcr	Conversione dal modello di colore rgb a YCbCr

Il prefisso "**im2**" sta per "**image to**", ed indica proprio che la funzione effettua una conversione "da immagine a " qualche altro tipo di file; discorso analogo per **rgb2** ecc...

Questo elenco si riferisce alle funzioni di conversione che riguardano le immagini e ovviamente NON è esaustivo (ci sono funzioni per i *frames* dei filmati, la scala di grigi, *colormodel* differenti, ...) e si consiglia di consultare l'**help online di Matlab**, soprattutto le sezioni "**See Also**" delle descrizioni dei comandi.

La seguente porzione di codice **carica un'immagine**, ne crea una **copia** passando l'immagine caricata in formato **uint8**, in **scala di grigi** e nel modello **YCbCr** e mostra le quattro immagini a video (nota: dovreste inserire un'immagine con nome **"prova.jpg"** nella vostra **directory di lavoro** o **Matlab** restituirà **errore (file non trovato)**; copiate quindi la porzione di codice che segue nella vostra **Command Window** e premete Invio):

```
>> immagine1 = imread('prova.jpg');  
>> immagine2 = uint8(immagine1);  
>> immagine3 = rgb2gray(immagine1);  
>> immagine4 = rgb2ycbcr(immagine1);  
  
>> subplot(2,2,1); imshow(immagine1); subplot(2,2,2);  
imshow(immagine2);  
  
>> subplot(2,2,3); imshow(immagine3); subplot(2,2,4);  
imshow(immagine4);
```

* * *

Le basi - Tutorial 12

Operazioni di base sulle immagini: funzioni del gruppo im

Alcune particolari **funzioni**, che iniziano in genere per **"im"** (o **"im2"**, per quelle di conversione da immagini ad altri tipi di dati, viste nel capitolo precedente), permettono di lavorare facilmente sulle immagini (**im** sta proprio per **image**).

Il seguente è un elenco - ovviamente non esaustivo - di alcune funzioni di base del **gruppo im**:

imadd	Somma tra loro due immagini o una costante (un valore scalare, come un double o un intero) ad un'immagine
imsubtract	Sottrae tra loro due immagini o una costante (un valore scalare, come un double o un intero) ad un'immagine
immultiply	Moltiplica due immagini tra loro o un'immagine per una costante (un valore scalare, come un double o un intero)
imdivide	Divide due immagini tra loro o un'immagine per una costante (un valore scalare, come un double o un intero)
imabsdiff	Restituisce la differenza, passata poi in valore assoluto, di due immagini
imcomplement	Restituisce il complemento (o: complementare) di un'immagine
imlincomb	Calcola una combinazione lineare di due o più immagini

Una funzione particolarmente interessante è poi **imfilter**, che permette di applicare alle immagini passate come parametri **filtri per i bordi, operazioni puntuali e filtri convolutivi**, tutti argomenti trattati nella **seconda parte** di questa raccolta di tutorial; la prima parte, relativa alle basi di Matlab, si chiude invece qui.

* * *

E.I.D. - Tutorial 1

Matlab per l'elaborazione delle immagini digitali: le basi

Questo tutorial vuole essere un'introduzione ai comandi di base per **manipolare delle immagini mediante Matlab**.

Per **caricare un'immagine** presa dal disco nell'area di lavoro di **Matlab**, bisogna utilizzare la funzione:

```
imread('nomefile.estensione');
```

Ovviamente, così facendo non depositeremo l'immagine da nessuna parte, per cui è necessario **creare una variabile**, ad esempio **'immagine'**, e scrivere:

```
immagine = imread('nomefile.estensione');
```

'immagine' diverrà quindi una matrice (potete averne conferma, e leggere alcuni dati, consultando il **Workspace**), di dimensioni **mxn**, con **m** numero di righe (ossia: **l'altezza, in pixel**, dell'immagine) ed **n** di colonne (la **larghezza, in pixel**, dell'immagine).

Ogni cella della matrice, richiamabile con:

```
immagine(x,y)
```

(con **x** e **y** coordinate del pixel nell'immagine) conterrà il **valore associato a quel pixel**, utile soprattutto se la vostra immagine è in scala di grigi in bianco e nero; le immagini a colori (**RGB**) o con **alpha (RGBA)** vengono memorizzate come matrici ***MxNxd***, dove ***d*** è il **numero di livelli**: 3 se RGB, 4 se RGB con Alpha (opacità).

Scrivendo:

```
immagine(x,y,2)
```

otterrete il valore dell'intensità DEL COLORE VERDE (canale 2, cioè G, verde), espresso generalmente con un valore **uint8** (cioè **unsigned int 8 bit**: da 0 a 255, per intenderci), del pixel di posizione *x* (coordinata verticale), *y* (coordinata orizzontale).

Per **memorizzare i valori di altezza e larghezza dell'immagine** nelle variabili **M** ed **N** rispettivamente, potete utilizzare il comando:

```
[M, N] = size(immagine) ;
```

Scrivendo semplicemente **M** o **N** nella finestra dei comandi, e premendo invio, vi verrà mostrato a video il valore dell'altezza o della larghezza in pixel dell'immagine.

Potete **creare una nuova variabile-immagine** a partire da un'altra, ad esempio per effettuare test diversi senza dover ricaricare il file, semplicemente con = :

```
immagine2 = immagine;
```

A questo punto, per effettuare '**operazioni puntuali**' (pixel per pixel, per intenderci) sull'immagine, potete 'scorrere' la stessa con, ad esempio, **uno o più cicli for**, per elaborare tutta l'immagine o parti di essa, o ancora tutti i canali o solo un canale-colore.

La porzione di **script** seguente illustra, ad esempio, **come rendere tutta l'immagine interamente bianca**:

```
for m=1:M  
for n=1:N  
immagine2(m,n) = 255;  
end;  
end;
```

E per visualizzare quanto fatto? Utilizzate il comando:

```
imshow([nomeVariabileImmagine]);  
imshow(immagine2)
```

Per salvare l'immagine così prodotta su disco, utilizzate la funzione:

```
imwrite(variabile, '[nomeDelFile]', '[formatoDelFile]');  
imwrite(immagine2, 'immagine2.jpg', 'jpg');
```

che salverà l'immagine nella directory dell'**area di lavoro di Matlab**.

[NOTA: Matlab non vi chiederà il permesso di sovrascrivere eventuali file omonimi, ma lo farà e basta, senza avvertirvi né prima né dopo averlo fatto!]

Potete anche **salvare parti o parti di canali dell'immagine**, semplicemente depositando queste sezioni in una nuova variabile e salvando quest'ultima su disco, utilizzando ad esempio uno script del genere, che ricopia un quadrato di dimensioni 30x30 pixel preso a partire dal vertice (101,301) (con 101 coordinata verticale) di immagine, *mettendo però a zero* (cioè annullandolo) *il contributo del canale verde (2) del colore originale*:

```
porzione = ones(30,30,3); % Crea una matrice di dimensioni 30x30x3  
porzione = uint8(porzione); % Rende la matrice una matrice di uint8  
(il tipo di dati di default di una matrice-immagine)  
for m=101:130  
    for n=301:330  
        porzione(m-100,n-300,1) = immagine(m,n,1);  
        porzione(m-100,n-300,2) = 0;  
        porzione(m-100,n-300,3) = immagine(m,n,3);  
    end;  
end;  
imwrite(porzione, 'porzione.jpg', 'jpg');
```

(ovviamente, per poter utilizzare questo script così come l'ho scritto, la vostra immagine dovrà essere più larga di 330 pixel e più alta di 130 pixel).

* * *

E.I.D. - Tutorial 2

FFT: trasformata di Fourier (e IFFT, l'inversa)

Questo script **carica un file immagine**, *'immagine.jpg'*, ne **calcola la trasformata di Fourier**, la “*shifta*” (sposta), ne ricava la **complessa coniugata** e da quest'ultima ricava l'**antitrasformata di Fourier**, mostrando a video il risultato.

Ecco lo **script**:

```
clear;
immagine = imread('immagine.jpg');

fourier = fft2(immagine); % Funzione per calcolare la Fast Fourier
Transform di un'immagine (2D)

fourier = fftshift(fourier); % Shift della trasformata
[M, N] = size(fourier);

complessaConiugata = zeros(M,N); % Creo una matrice di zeri della
dimensione della trasformata di Fourier; tale matrice conterrà la
complessa coniugata

for x=1:M
    for y=1:N
        complessaConiugata(x,y) = fourier(abs(x-(M+1)),
            abs(y-(N+1)));
    end;
end;
```

```
% A questo punto, calcolo la trasformata inversa shiftata della
complessa coniugata e la mostro a video:
ifft = ifftshift(complessaConiugata);

ifft = real(ifft2(ifft)); % ifft2 produce una matrice di numeri
complessi; noi possiamo considerare e mostrare a video la parte reale,
ignorando la parte immaginaria (la teoria che sta alla base di questa
considerazione esula da questa trattazione)

imshow(ifft);
```

Da notare che è possibile **visualizzare** a video anche la **trasformata di Fourier** di un'immagine:
basta scrivere ad esempio

```
immagine = imread('immagine.jpg');
fourier = fft2(immagine);
imshow(fourier);
```

* * *

E.I.D. - Tutorial 3

Filtri gaussiani LP e HP

Questo script apre un file immagine, 'immagine.jpg', e mostra a video due immagini, derivanti dall'applicazione dei filtri LP e HP (low-pass e high-pass, cioè passa-basso e passa-alto) gaussiani.

L'operazione viene effettuata nel **dominio di Fourier**, per cui è necessario ricavare lo **spettro di Fourier dell'immagine tramite *fft* (fast fourier transform)**, creare i **filtri gaussiani** per tale spettro, applicarli allo spettro dell'immagine e riconvertire il tutto passando, mediante **l'antitrasformata**, al dominio spaziale.

[NOTA IMPORTANTE: utilizzare, con questo script, solo immagini in scala di grigi a 8 bit, o Matlab restituirà un errore per via delle dimensioni delle matrici, che non coincideranno.]

Ecco lo script:

```
% Operazioni di base: caricamento dell'immagine, passaggio al dominio
di Fourier
clear;
immagine = imread('immagine.jpg');
fourier = fft2(immagine);
[M,N] = size(immagine);
```

```
% Le istruzioni che seguono servono a costruire una gaussiana
u=0:(M-1);
v=0:(N-1);
idx = find(u>M/2);
u(idx) = u(idx)-M;
idy = find(v>N/2);
v(idy) = v(idy)-N;
[V,U] = meshgrid(v,u);

% Passiamo quindi alla definizione e all'applicazione dei filtri
gaussiani, partendo dall'equazione della stessa (la teoria che
descrive tali filtri non viene qui trattata):
d0 = 25; % Imposto 25 come frequenza di taglio dei filtri
D = sqrt(U.^2+V.^2);
H_LP_GAUSS = exp(-(D.^2)./(2*(d0^2)));
G1 = H_LP_GAUSS.*fourier;
passaBasso = real(ifft2(G1));
H_HP_GAUSS=1-exp(-(D.^2)./(2*(d0^2)));
G2 = H_HP_GAUSS.*fourier;
passaAlto = real(ifft2(G2));

% Conversione a uint8 delle immagini ottenute
passaBasso = uint8(passaBasso);
passaAlto = uint8(passaAlto);

% Mostro a video le tre immagini: originale, con passaBasso, con
passaAlto
subplot(1,3,1); imshow(immagine); subplot(1,3,2);
imshow(passaBasso); subplot(1,3,3); imshow(passaAlto); .
```

* * *

E.I.D. - Tutorial 4

imfilter: media aritmetica e geometrica, mediano, midpoint, ...

In questo articolo mostrerò come definire ed applicare, mediante **script in Matlab**, alcuni **filtri per immagini digitali**: media aritmetica e geometrica, **mediano**, **mid-point** e **controarmonico**.

La teoria che sta alla base di queste operazioni *non* verrà trattata in questo tutorial: qui ci limitiamo a vedere *come realizzare* certe operazioni in Matlab.

*[NOTA IMPORTANTE: utilizzare, con questo script, solo **immagini in scala di grigi a 8 bit**, o **Matlab** restituirà un errore per via delle dimensioni delle matrici, che non coincideranno.]*

Questo script **non va ricopiato per intero in una volta**, ma eseguito 'passo-passo', ogni volta che trovate una riga contenente una o più istruzioni **imshow**: in effetti, con questo script vengono presentati più filtri, con **dimensioni di maschere differenti**, che mostrati tutti insieme genererebbero solo confusione...

*[Altra nota importante: per eseguire tutte le porzioni degli **script** in maniera corretta dovrete procurarmi i file di libreria **gmean** e **charmean** (se non sono già presenti nel vostro pacchetto); li trovate comunque gratuitamente, online.]*

Ecco lo **script** (da eseguire, come detto precedentemente, “passo dopo passo”):

```
>> clear;  
>> immagine = zeros(234,234);  
>> for i=1:9
```

```
immagine(12:224, ((17*i+(7*(i-1))):(17*i+(7*(i-1)+7)))) = 1;
end;
>> imshow(immagine);

% Applico i filtri di media aritmetica
>> w1 = fspecial('average', [3,3]);
>> arit3x3 = imfilter(immagine,w1);
>> w2 = fspecial('average', [5,5]);
>> arit5x5 = imfilter(immagine,w2);
>> w3 = fspecial('average', [7,7]);
>> arit7x7 = imfilter(immagine,w3);
>> subplot(2,2,1); imshow(immagine); subplot(2,2,2); imshow(arit3x3);
subplot(2,2,3); imshow(arit5x5); subplot(2,2,4); imshow(arit7x7);

% Applico i filtri di media geometrica
>> mediaGeometrica3x3 = gmean(immagine, 3,3);
>> mediaGeometrica5x5 = gmean(immagine, 5,5);
>> mediaGeometrica7x7 = gmean(immagine, 7,7);
>> subplot(2,2,1); imshow(immagine); subplot(2,2,2);
imshow(mediaGeometrica3x3); subplot(2,2,3);
imshow(mediaGeometrica5x5); subplot(2,2,4);
imshow(mediaGeometrica7x7);

% Applico il filtro mediano
>> mediano3x3 = medfilt2(immagine, [3,3]);
>> mediano5x5 = medfilt2(immagine, [5,5]);
>> mediano7x7 = medfilt2(immagine, [7,7]);
>> subplot(2,2,1); imshow(immagine); subplot(2,2,2);
imshow(mediano3x3); subplot(2,2,3); imshow(mediano5x5);
subplot(2,2,4); imshow(mediano7x7);

% Applico il filtro mid-point
>> minimo3x3 = ordfilt2(immagine, 1, ones(3,3));
>> massimo3x3 = ordfilt2(immagine, 9, ones(3,3));
>> midpoint3x3 = 0.5 .* (minimo3x3 + massimo3x3);
>> minimo5x5 = ordfilt2(immagine, 1, ones(5,5));
>> massimo5x5 = ordfilt2(immagine, 25, ones(5,5));
>> midpoint5x5 = 0.5 .* (minimo5x5 + massimo5x5);
>> minimo7x7 = ordfilt2(immagine, 1, ones(7,7));
```

```
>> massimo7x7 = ordfilt2(immagine, 49, ones(7,7));
>> midpoint7x7 = 0.5 .* (minimo7x7 + massimo7x7);
>> subplot(2,2,1); imshow(immagine); subplot(2,2,2);
imshow(midpoint3x3); subplot(2,2,3); imshow(midpoint5x5);
subplot(2,2,4); imshow(midpoint7x7);

% Applico il filtro di media controarmonica con Q=1
>> controarmonico3x3 = charmean(immagine, 3,3, 1);
>> controarmonico5x5 = charmean(immagine, 5,5, 1);
>> controarmonico7x7 = charmean(immagine, 7,7, 1);
>> subplot(2,2,1); imshow(immagine); subplot(2,2,2);
imshow(controarmonico3x3); subplot(2,2,3); imshow(controarmonico5x5);
subplot(2,2,4); imshow(controarmonico7x7);
```

* * *

E.I.D. - Tutorial 5

imnoise e il rumore impulsivo

In questo tutorial vedremo come utilizzare la funzione di libreria *imnoise* per **aggiungere del rumore ad un'immagine**. Aggiungere del rumore “artificiale” ad un'immagine è utile per **valutare l'efficienza** degli algoritmi di “pulizia / restauro” delle immagini digitali.

Lo **script** qui presente, in particolare, aggiunge del **rumore impulsivo** (*'salt-and-pepper'*, sale e pepe) ad un'immagine caricata da file (*'immagine.jpg'*).

Quello impulsivo non è l'unico tipo di rumore disponibile: abbiamo, ad esempio, *'gaussian'*, *'localvar'*, *'poisson'* e *'speckle'*, ciascuno con i propri parametri; per maggiori informazioni, digitare:

```
help imnoise
```

nella **Command Window** di **Matlab**.

Ecco lo **script**:

```
clear;
immagineInput = imread('immagine.jpg');
immagineConDegrado = imnoise(immagineInput, 'salt & pepper', 0.05);
immagineConDegrado = uint8(immagineConDegrado);
subplot(1,2,1); imshow(immagineInput); subplot(1,2,2);
imshow(immagineConDegrado);
```

* * *

E.I.D. - Tutorial 6

Il filtro di “degrado atmosferico” (Hufnagel-Stanley)

In questo tutorial vedremo come implementare in **Matlab** un **filtro di degrado**, utilizzato per esaminare la qualità dei filtri di restauro delle immagini digitali: il **filtro di degrado 'atmosferico' di Hufnagel-Stanley**.

Lo script apre un file immagine, **'immagine.jpg'**, ne ricava lo **spettro di Fourier (con la FFT)** e a questo applica il **filtro di Hufnagel e Stanley**, costituito da una matrice (il valore di ogni cella è definito da una equazione che tiene conto della posizione della stessa; tale equazione, che è appunto quella del filtro, viene implementata in **Matlab** all'interno di due **for** annidati); del risultato viene fatta, quindi, l'**antitrasformata**, che viene mostrata a video.

[NOTA IMPORTANTE: utilizzare, con questo script, solo immagini in scala di grigi a 8 bit, o Matlab restituirà un errore per via delle dimensioni delle matrici, che non coincideranno.]

Ecco il **codice dello script**:

```
clear;
immagine = imread('immagine.jpg');
fourier = fft2(immagine);
[M,N] = size(immagine);
filtro = zeros(M,N);
```

*k = 0.0001; % Coefficiente da modificare per impostare la turbolenza;
range consigliato: 0.0025 per turbolenza forte, 0.00025 per debole.*

```
for x=1:M  
    for y=1:N  
        filtro(x,y) = exp(-k .* (x^2+y^2)^(5/6));  
    end;  
end;
```

```
filtrata = filtro .* fourier;
```

```
antiFourier = real(ifft2(filtrata));  
antiFourier = uint8(antiFourier);
```

```
subplot(1,2,1); imshow(immagine); subplot(1,2,2); imshow(antiFourier);
```

* * *

E.I.D. - Tutorial 7

Il filtro di restauro Butterworth

In questo articolo vedremo come implementare, con uno **script Matlab**, un **filtro di Butterworth per il restauro delle immagini digitali** con degrado.

Per simulare del **degrado** su di un'immagine potete utilizzare il **filtro di Hufnagel-Stanley**, presentato nel tutorial precedente.

Lo script apre un file immagine, '**immagine.jpg**', ne ricava lo **spettro di Fourier (con la FFT)**, costruisce il **filtro di Butterworth** (viene implementata l'equazione dello stesso) e lo **applica allo spettro di Fourier dell'immagine**; del risultato viene fatta, quindi, l'**antitrasformata**, che viene mostrata infine a video.

L'applicazione del **filtro** avviene, quindi, **nello spettro di Fourier**.

[NOTA IMPORTANTE: utilizzare, con questo script, solo immagini in scala di grigi a 8 bit, o Matlab restituirà un errore per via delle dimensioni delle matrici, che non coincideranno.]

Il **codice di Butterworth** utilizza due parametri: **ordine** e **raggio**; nello script implementato qui di seguito, sono stati scelti i valori 10 e 70 per tali parametri.

Come per gli altri tutorial di questa sezione, la teoria che sta alla base di queste operazioni *non* verrà trattata in questo tutorial: qui ci limitiamo a vedere *come realizzare* certe operazioni in Matlab.

Ecco il **codice** dello script:

```
% Istruzioni di base
clear;
immagine = imread('immagine.jpg');

fourier = fft2(immagine);
[M,N] = size(immagine);

% Le istruzioni che seguono servono a costruire il filtro di Butterworth
ordine = 10; % Uno dei due parametri da impostare esplicitamente
raggio = 70; % Il secondo parametro da impostare esplicitamente
u = 0:(M-1);
v = 0:(N-1);
idx = find(u>M/2);
u(idx) = u(idx)-M;
idy = find(v>N/2);
v(idy) = v(idy) - N;
[V,U] = meshgrid(v,u);
D = sqrt(U.^2 + V.^2);

butterworthFilter = 1 ./ (1+(D./raggio).^(2.* ordine));

% Applico il filtro alla fft dell'immagine, ricavo l'antitrasformata e
la mostro a video
butterworthFilter = fourier .* butterworthFilter;

antitrasformata = real(ifft2(butterworthFilter));
antitrasformata = uint8(antitrasformata);

subplot(1,2,1); imshow(immagine); subplot(1,2,2);
imshow(antitrasformata);
```

* * *

E.I.D. - Tutorial 8

Elementi strutturali e operatori morfologici

Lo **script** qui presentato modella, con degli **operatori morfologici**, un'immagine di input (creata, in questo caso, all'interno dello script).

Verrà fatto uso, quindi, di **'operatori morfologici' messi a disposizione da Matlab** (ne verranno mostrati solo tre: *imerode*, *imdilate* e *imclose*; sono disponibili altri strumenti, per i quali si rimanda, però, alla **documentazione di Matlab**) e di elementi strutturali per la modellazione, creati mediante la funzione di libreria *strel* (per maggiori informazioni, digitate *'help strel'* , senza apici, nella **Command View di Matlab**).

L'analisi della **teoria** che sta alla base della modifica delle immagini digitali mediante elementi strutturali e operatori morfologici esula da questa trattazione.

[NOTA IMPORTANTE: utilizzare, con questo script, solo immagini in scala di grigi a 8 bit, o Matlab restituirà un errore per via delle dimensioni delle matrici, che non coincideranno.]

Ecco il **codice dello script**:

```
% Creo un'immagine in bianco e nero all'interno di Matlab
clear;
immagine = zeros(250,250);
immagine(50:150, 50:100) = 1;
immagine(150:200, 50:200) = 1;
immagine(50:150, 150:200) = 1;
```

% Prima applicazione ('caso A') di operatori morfologici, in realtà doppia: un passo di erosione con elemento strutturante di forma rettangolare seguito da un passo di dilatazione con elemento strutturante di forma circolare

```
matricelc = ones(52, 51);  
elementoStrutturante1a = strel('arbitrary', matricelc);  
casoAprimoPasso = imerode(immagine, elementoStrutturante1a);  
elementoStrutturante2a = strel('disk', 15);  
casoAsecondoPasso = imdilate(casoAprimoPasso,  
    elementoStrutturante2a);
```

% Seconda applicazione ('caso B') di operatori morfologici, anche questa doppia: dilatazione con elemento circolare seguita da chiusura con elemento circolare

```
elementoStrutturante1b = strel('disk', 10);  
casoBprimoPasso = imdilate(immagine, elementoStrutturante1b);  
elementoStrutturante2b = strel('disk', 15);  
casoBsecondoPasso = imclose(casoBprimoPasso,  
    elementoStrutturante2b);
```

% Mostro a video l'immagine originale e i risultati ottenuti, mostrando anche i passaggi intermedi (primo e secondo passo, per entrambi i casi, un caso per riga)

```
subplot(2,3,1); imshow(immagine);  
subplot(2,3,2); imshow(casoAprimoPasso);  
subplot(2,3,3); imshow(casoAsecondoPasso);  
subplot(2,3,4); imshow(immagine);  
subplot(2,3,5); imshow(casoBprimoPasso);  
subplot(2,3,6); imshow(casoBsecondoPasso);
```

* * *

E.I.D. - Tutorial 9

Immagine 'sintetica' con figure da salvare su disco

In questo tutorial vedremo come creare, in **Matlab**, un'immagine 'sintetica', partendo da una matrice, con **figure geometriche posizionate casualmente** all'interno della stessa, per poi **salvare** il tutto su disco.

Le **figure geometriche** verranno create con delle **matrici** (per creare il triangolo, verrà creato un quadrato e, dopo, sarà necessario utilizzare **due cicli for annidati** per mettere a 0 la parte 'complementare' a quella del triangolo vero e proprio).

Le **coordinate dei vertici-origine** delle figure (i vertici 'in alto a sinistra') verranno generate con la **funzione rand**, mantenendo comunque i valori all'interno dell'area di lavoro (che poi equivale alla dimensione, in pixel, dell'immagine finale, per intenderci).

Lo **script** mostra a video **l'immagine e la salva su disco**, con nome file '**output.jpg**'; **attenzione:** se nella cartella di lavoro è presente un file '**output.jpg**', questo script lo **sovrascriverà** senza avvisarvi (né prima né dopo)!

Questo tutorial sta alla base di quello dedicato alla creazione e al salvataggio su disco di FILMATI 'sintetici' (creati interamente in Matlab, mediante matrici, senza dati in input); in quel caso, comunque, non avremo a che fare con immagini ma con **FRAMES**, come vedremo.

Il codice dello script si trova nella pagina seguente.

```
% Impostazioni iniziali: dimensioni (M per l'altezza, N per la larghezza)
in pixel dell'area di lavoro-immagine di output e 'disegno' delle figure
geometriche (un rettangolo e un triangolo)
clear;
M = 160;
N = 160;
areaLavoro = zeros(M,N,3); % Figure a 3 dimensioni (RGB)
rettangolo = ones(15,10,3);
triangolo = ones(10,10,3);
for i=1:10
    for j = (i+1):10
        triangolo(i,j,:)=0;
    end;
end;
% Genero le coordinate iniziali dei vertici in alto a sinistra delle figure
per posizionarle nell'area di lavoro
posInizialeTriangoloX = floor(rand(1) * M);
posInizialeTriangoloY = floor(rand(1) * N);
posInizialeRettangoloX = floor(rand(1) * M);
posInizialeRettangoloY = floor(rand(1) * N);

% Disegno il triangolo e il rettangolo nell'area di lavoro
for i=1:10
    for j= 1:10
        areaLavoro(posInizialeTriangoloX + i, posInizialeTriangoloY + j) =
            triangolo(i,j); % Posiziono il triangolo nell'area di lavoro
    end;
end;
for i=1:15
    for j= 1:10
        areaLavoro(posInizialeRettangoloX + i, posInizialeRettangoloY + j) =
            rettangolo(i,j); % Posiziono il rettangolo nell'area di lavoro
    end;
end;
% Mostro a video l'immagine e la salvo su disco, nome file: 'output.jpg'
imshow(areaLavoro);
imwrite(areaLavoro, 'output.jpg', 'jpg');

*      *      *
```

E.I.D. - Tutorial 10

Animazione casuale e FRAMES

In questo tutorial vedremo come creare, in **Matlab**, un'animazione 'sintetica', partendo da un'immagine anch'essa creata al volo, senza input, contenente **figure geometriche posizionate casualmente** all'interno della stessa, creando dei **FRAMES** da concatenare, per poi salvare il tutto su disco come **FILMATO**.

Questo tutorial segue direttamente quello dedicato alla generazione di immagini sintetiche e al loro salvataggio mediante Matlab; la prima parte dello script è uguale in tutto e per tutto, salvo per le ultime 3 righe, allo script presentato nel tutorial precedente, per cui non verrà ridiscussa: ci concentreremo, invece, sulla seconda parte.

Ad ogni frame dell'animazione (ciclo *for* che va da 1 a 25, ma potete impostare un numero di **frames** o un *frame rate* differenti, ovviamente) viene aggiunto o sottratto un **vettore di spostamento casuale alle figure geometriche**, che vengono quindi ridisegnate di volta in volta, nelle nuove posizioni, nell'area di lavoro.

Dall'immagine dell'area di lavoro si deve, poi, ricavare un oggetto **FRAME**, da posizionare all'interno del filmato; il filmato vero e proprio, comunque, verrà creato alla fine, fuori dal ciclo **for**, e lì potremo impostare il numero di visualizzazioni da effettuare a video e il **frame rate**.

La variabile **'mappa'** si riferisce alla mappa colori creata dalla **funzione gray2ind** per indicizzare, come suggerisce il nome della funzione stessa, l'immagine (passaggio **da immagine a FRAME**).

Per memorizzare l'animazione su file, è necessario creare un **oggetto avifile** ed aggiungergli tutti i frames creati di volta in volta, memorizzati nel **vettore dei frame "frames"**.

Per maggiori informazioni sui comandi *rgb2gray*, *gray2ind*, *movie*, *avifile*, ecc..., digitare **'help [nomeistruzione]'** nella **Command Window di Matlab**.

Lo **script** mostra a video il filmato e lo salva su disco, con nome file **'filmato.avi'**; **attenzione:** se nella cartella di lavoro è presente un file **'filmato.avi'**, questo script lo sovrascriverà senza avvisarvi (né prima né dopo)!

Qui di seguito, il codice dello **script**.

```
% PRIMA PARTE --- Identica a quella dell'articolo precedente
%
% Impostazioni iniziali: dimensioni (M per l'altezza, N per la larghezza)
% in pixel dell'area di lavoro-immagine di output e 'disegno' delle figure
% geometriche (un rettangolo e un triangolo)
clear;
M = 160;
N = 160;
areaLavoro = zeros(M,N,3); % Figure a 3 dimensioni (RGB)
rettangolo = ones(15,10,3);
triangolo = ones(10,10,3);
for i=1:10
    for j = (i+1):10
        triangolo(i,j,:)=0;
    end;
end;

% Genero le coordinate iniziali dei vertici in alto a sinistra delle figure
% per posizionarle nell'area di lavoro
posInizialeTriangoloX = floor(rand(1) * M);
posInizialeTriangoloY = floor(rand(1) * N);
posInizialeRettangoloX = floor(rand(1) * M);
posInizialeRettangoloY = floor(rand(1) * N);
```

```
% Disegno il triangolo e il rettangolo nell'area di lavoro
for i=1:10
    for j= 1:10
        areaLavoro(posInizialeTriangoloX + i, posInizialeTriangoloY + j) =
            triangolo(i,j); % Posiziono il triangolo nell'area di lavoro
    end;
end;
for i=1:15
    for j= 1:10
        areaLavoro(posInizialeRettangoloX + i, posInizialeRettangoloY + j) =
            rettangolo(i,j); % Posiziono il rettangolo nell'area di lavoro
    end;
end;
```

% NON mostro a video e NON salvo l'immagine dell'area di lavoro ottenuta fino a questo punto...

```
% SECONDA PARTE: genero un'animazione. Ad ogni frame, aggiungo o sottraggo
un vettore spostamento casuale alla posizione dell'oggetto nel frame
precedente
for n=1:25
    areaLavoro = zeros(M,N,3); % Ripulisco l'area di lavoro, per "disegnarle
    sopra" il frame nella nuova posizione
```

```
% Applico, alla posizione del triangolo e del rettangolo del frame
precedente, uno spostamento casuale, positivo o negativo, lungo x e lungo y
spostamentoCasuale = floor(rand(1)*10) - floor(rand(1)*10);
if(((posInizialeTriangoloX + spostamentoCasuale) > 0) |
    ((posInizialeTriangoloX+10+spostamentoCasuale)<M-1))
    posInizialeTriangoloX = posInizialeTriangoloX + spostamentoCasuale;
end;
```

```
spostamentoCasuale = floor(rand(1)*10) - floor(rand(1)*10);
```

```
if(((posInizialeTriangoloY + spostamentoCasuale) > 0) |
    ((posInizialeTriangoloY+10+spostamentoCasuale)<N-1))
```

```
posInizialeTriangoloY = posInizialeTriangoloY + spostamentoCasuale;
end;

spostamentoCasuale = floor(rand(1)*10) - floor(rand(1)*10);

if(((posInizialeRettangoloX + spostamentoCasuale) > 0) |
((posInizialeRettangoloX+10+spostamentoCasuale)<M-1))
posInizialeRettangoloX = posInizialeRettangoloX + spostamentoCasuale;
end;

spostamentoCasuale = floor(rand(1)*10) - floor(rand(1)*10);

if(((posInizialeRettangoloY + spostamentoCasuale) > 0) |
((posInizialeRettangoloY+15+spostamentoCasuale)<N-1))
posInizialeRettangoloY = posInizialeRettangoloY + spostamentoCasuale;
end;

% Disegno il triangolo e il rettangolo, con le nuove posizioni, nell'area
di lavoro
for i=1:10
for j= 1:10
areaLavoro(posInizialeTriangoloX + i, posInizialeTriangoloY + j) =
triangolo(i,j); % Posiziono il triangolo nell'area di lavoro
end;
end;

for i=1:15
for j= 1:10
areaLavoro(posInizialeRettangoloX + i, posInizialeRettangoloY + j) =
rettangolo(i,j); % Posiziono il rettangolo nell'area di lavoro
end;
end;

% Dall'immagine ricavo il frame, e lo posiziono all'interno del "filmato"
(per il momento, "frames" è solo un vettore di frame, alla fine del for
creerò il vero filmato da visualizzare a schermo)
areaLavoro = rgb2gray(areaLavoro);
```

```
[frameCorrente, mappa] = gray2ind(areaLavoro,256);  
frames(n) = im2frame(frameCorrente, mappa);  
end;
```

```
% Dal vettore dei frame, ricavo il "movie" da mostrare a video  
% Il secondo parametro indica di eseguire la visualizzazione 1 volta, il  
terzo indica di visualizzare 1 frame al secondo (per poterlo analizzare  
meglio)  
filmato = movie(frames, 1,1);
```

```
% Per memorizzare l'animazione su file, devo creare un oggetto avifile ed  
aggiungergli tutti i frames creati di volta in volta, memorizzati nel  
vettore dei frame "frames":  
filmatoDaSalvareSuDisco = avifile('filmato.avi')  
filmatoDaSalvareSuDisco.Compression='None'; % Sono disponibili altri tipi  
di compressione; digitate help avifile nella Command Window per maggiori  
informazioni
```

```
for n=1:25  
filmatoDaSalvareSuDisco = addframe(filmatoDaSalvareSuDisco, frames(n));  
end;
```

```
filmatoDaSalvareSuDisco = close(filmatoDaSalvareSuDisco);
```

* * *

E.I.D. - Tutorial 11

Realizzare un filmato AVI da una sequenza di immagini

A volte è necessario dover **creare un filmato avi a partire da una sequenza di immagini jpg**, ad esempio immagini ottenute da grafici o processi di elaborazione di immagini o filmati, magari realizzati con lo stesso **Matlab**.

Possiamo effettuare tale operazione con lo script trattato in questo tutorial.

Si tratta di uno **script** che, oltre a mostrare alcune chiamate strettamente necessarie per **creare i filmati**, contiene anche delle istruzioni sulla **creazione di stringhe**, per cui può essere utile anche in tal senso.

Eccolo:

```
mov = avifile('output.avi');
estensione = 'b.JPG';
for i=0:239
p = double(i);
p = int2str(p);
nomefile = strcat(p, estensione);
immagine = imread(nomefile);
f = im2frame(immagine);
mov = addframe(mov, f);
end;
mov = close(mov);
```

Esaminiamolo passo per passo.

Con la prima istruzione **apriamo uno stream di output**, specificando che intendiamo scrivere **un file di tipo avi (avifile)**, cioè un **video**, da memorizzare con nome **'output.avi'**. Il percorso è dato dal nostro **workspace**.

Tale stream verrà "memorizzato" nella **variabile "mov"**.

La seconda istruzione riguarda la **"costruzione" del suffisso e dell'estensione** per i nomi dei file di input. Nel mio caso, infatti, i nomi dei file di input erano numerati (da 0 a 239) e finivano tutti in **"b"**, con estensione **"JPG"**.

Memorizzo quindi una stringa **"b.JPG"**, da utilizzare in seguito.

Il **ciclo for**, in questo caso, va da 0 a 239 perché, come ho detto, questo era il mio caso... ovviamente dovreste modificare questa sezione in base alle vostre esigenze (es.: una collezione di nomi di file, o un range numerico differente).

Le prime tre istruzioni contenute nel **for** non riguardano strettamente **la creazione del file avi**, ma la definizione del nome del file immagine da aprire e aggiungere al filmato.

Con ***p=double(i)*** effettuo una promozione esplicita di ***i*** a **double** e memorizzo tale valore in ***p***.

Questo **casting** mi serve per poter utilizzare il metodo ***int2str***, che trasforma l'argomento (un **double**) in una **stringa**. Nel mio caso, tale stringa è di nuovo ***p*** (Matlab mi consente di **cambiare il tipo** di ***p*** a runtime).

Ho infatti bisogno di una stringa per poter concatenare l'indice del ciclo, numerico, e l'estensione precedentemente dichiarata.

Adesso che ho questi ingredienti, li unisco con ***strcat(p, estensione)***, ossia **"strings concatenation"**: concatenazione di stringhe che, come è facilmente intuibile, "attacca in coda" al primo parametro il secondo.

Memorizzo il nome del file di input così generato nella **variabile "nomefile"**.

A questo punto avviene la lettura del file di input: mediante l'istruzione ***imread*** (image read, lettura immagine) carichiamo il file di nome "***nomefile***" (la stringa costruita un attimo prima) e ne depositiamo il contenuto nella **variabile** "***immagine***".

Per **creare un filmato**, comunque, non possiamo aggiungere direttamente le immagini così come sono.

I filmati sono costituiti da FRAMES. Durante la riproduzione del filmato, viene mostrato un certo numero di **frames al secondo** (di default, 15), quindi attenzione a quanti ne inserite (a volte potrebbero essere necessarie delle ripetizioni).

Il numero di frames al secondo (***fps***) e altri parametri (come la **compressione** o la **qualità**) possono essere impostati esplicitamente durante la creazione del nostro ***avifile*** (prima istruzione, in questo **script**) o anche successivamente; per maggiori dettagli, vedere la pagina riguardante la funzione ***avifile*** su **MathWorks**.

Con ***im2frame***, quindi, effettuiamo una **conversione da immagine a frame**, come suggerisce il nome della funzione, a partire dall'immagine "***immagine***", depositando il risultato nella **variabile** ***f***.

A questo punto abbiamo il nostro **frame**, che aggiungiamo in coda al filmato con ***mov = addframe(mov, f)***.

Il ciclo **for** viene ripetuto per inserire tutti i file immagine **in coda al frame**.

Come ultima istruzione abbiamo ***close(mov)***, per chiudere lo stream di scrittura sul file di output.

* * *

Extra - Tutorial 1

Risoluzione delle equazioni differenziali con ODE (con esempio)

Per chi ha la necessità di **risolvere equazioni differenziali**, **ODE** e loro **sistemi**, **Matlab** mette a disposizione alcune funzioni di libreria davvero utili, a seconda delle esigenze:

Ode45	Problemi non-stiff, ordine di accuratezza medio
Ode23	Problemi non-stiff, ordine di accuratezza basso
Ode113	Problemi non-stiff, ordine di accuratezza variabile (basso-alto)
Ode15s	Problemi stiff, ordine di accuratezza variabile (basso-medio)
Ode23s	Problemi stiff, ordine di accuratezza basso
Ode23t	Problemi "moderatamente stiff", ordine di accuratezza basso
Ode23tb	Problemi stiff, ordine di accuratezza basso

Per utilizzare tali **funzioni**, è necessario definire l'**equazione differenziale** (o le equazioni differenziali, nel caso di un **sistema**) in un file **.m**, ove andranno definiti eventualmente anche altri parametri necessari per il calcolo, mentre bisognerà predisporre, nell'**area di lavoro** di **Matlab**, un **array** per l'**output**.

L'esempio seguente mostra come impostare il **sistema di ODE** per il **problema di Keplero modificato** (un tipo particolare di **sistema hamiltoniano**), composto da **quattro equazioni** con **quattro incognite**.

Ecco l'**equazione** del **sistema hamiltoniano H**:

$$H(\vec{q}, \vec{p}) = \frac{p_1^2 + p_2^2}{2} - \frac{1}{r} - \frac{\alpha}{2r^3}$$

dove **r** è la radice quadrata della somma dei quadrati di q_1 e q_2 .

Il valore **alpha** va impostato a piacere; nel nostro caso, abbiamo scelto 0.01.

Le **quattro incognite** sono quindi p_1 , p_2 , q_1 , q_2 ; scrivendo $\vec{p}$ ci riferiremo al **"vettore" di variabili p**, ossia in questo caso p_1 e p_2 ; discorso analogo va fatto per q .

Il **sistema di ODE** per tale **problema** è il seguente:

$$\begin{aligned}\vec{q}' &= \vec{p} \\ \vec{p}' &= -\left(\frac{1}{r^3} + \frac{3\alpha}{2r^5}\right)\vec{q}\end{aligned}$$

In questo caso, in un file **.m** a parte (**funzioneKeplero.m**, in questo caso) si crea l'**array** contenente le **equazioni del sistema**; il contenuto di tale **file** è il seguente:

```
%File funzioneKeplero.m
function funzione = funzioneKeplero(t, parametri) %parametri(1) =
pliniziale; parametri(3) = qliniziale

alpha = 0.01; % Non posso specificarlo fuori e passarlo come quinto
parametro: Matlab accetta solo un array di lunghezza 4 come secondo
argomento per ode45

r = sqrt(parametri(3).^2 + parametri(4).^2);

coeff = - ((1 ./ (r.^3)) + ((3 .* alpha) ./ (2 .* r.^5)));

funzione = [coeff .* parametri(3); coeff .* parametri(4);

parametri(1); parametri(2)]; %eq (nell'ordine) q1, q2, p1, p2
```

Adesso, nella **Command Window** di **Matlab**, bisogna fare in modo che una funzione della suite **ODE** del programma elabori, dato l'**intervallo di tempo** e il **passo**, tale **sistema**, depositando l'**output** (i valori delle quattro variabili p_1 , p_2 , q_1 , q_2) **nell'array** "output", appunto; in realtà,

output diverrà una **matrice**: ad ogni **riga**, si avranno i **valori delle quattro variabili** per un dato **istante di tempo** (ricordiamoci che, dato ad esempio l'intervallo $[0, 500]$, impostando il passo $h = 0.1$ avremo, in realtà, $(500-0) * (1/h) = 5000$ **righe-istanti di tempo**, per cui richiamando la riga 3100 di output, per esempio, staremo esaminando l'istante temporale 310, e così via).

Ecco uno **script** che potete copiare nella **Command Window di Matlab** (ovviamente, dovrete avere il file ***funzioneKeplero.m*** nella vostra area di lavoro) per ottenere il **grafico**, come vedrete a video, il "diagramma di fase" delle variabili q_1 e q_2 per tale **sistema** nell'**intervallo** $[0,500]$ con passo 0.1 con il metodo **MidPoint (ODE23)** e con **Runge-Kutta del quarto ordine (ODE45)**.

```
>> %Istruzioni da eseguire nella Command Window di Matlab
>> a = 0;
>> b = 500;
>> h = 0.1;
>> intervallo = a:h:b;

>> beta = 0.6;
>> qliniziale = 1- beta;
>> q2iniziale = 0;
>> pliniziale = 0;
>> p2iniziale = sqrt((1+beta)/(1-beta));

>> parametri = [pliniziale, p2iniziale, qliniziale, q2iniziale];

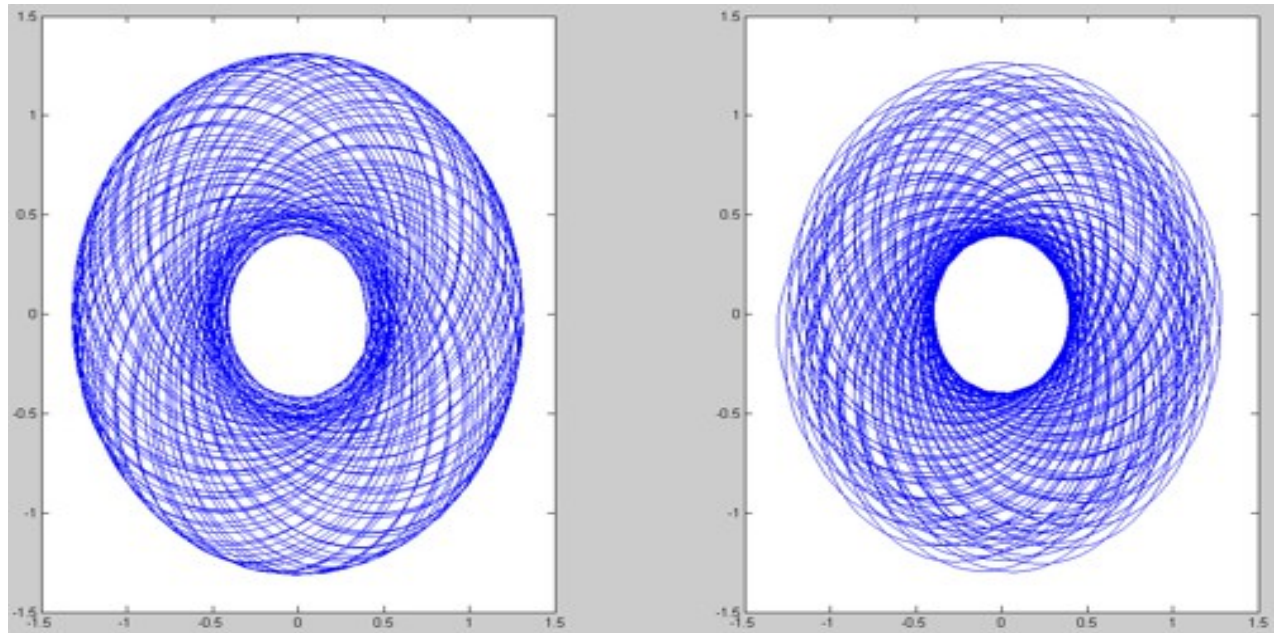
>> %MidPoint (ODE23) con opzioni di tolleranza di default
>> [t, output] = ode23('funzioneKeplero', intervallo, parametri);

>> %RK4 (ODE45) con opzioni tolleranza impostate esplicitamente
mediante ODESET
>> opzioniTolleranza = ODESET('RelTol', 1e-4, 'AbsTol', 1e-6); %Valori
di default: 1e-3 e 1e-6, ossia 0.01% e 0.00001%
>> [t, output2] = ode45('funzioneKeplero', intervallo, parametri,
opzioniTolleranza);

>> subplot(1,2,1);
>> xlabel('MidPoint esplicito');
>> plot(output(:,3), output(:,4));
```

```
>> subplot(1,2,2);  
>> xlabel('RK4');  
>> plot(output2(:,3), output2(:,4));  
>>
```

Ecco l'immagine che otterrete a video:



Non vi sarà sfuggita la **riga di codice**:

```
>> opzioniTolleranza = ODESET('RelTol', 1e-4, 'AbsTol', 1e-6); %Valori  
di default: 1e-3 e 1e-6, ossia 0.01% e 0.00001%
```

utilizzata per impostare i **valori di tolleranza relativa ed assoluta** per la funzione **ODE** da utilizzare.

In **Matlab** infatti è possibile definire due tipi di "**tolleranze**" nella risoluzione delle ODE: **relativa** ed **assoluta**.

La **relazione** tra tali valori e l'**errore** ad ogni **passo** è la seguente (dalla documentazione di **Matlab**):

$$e(i) \leq \max(\text{RelTol} * \text{abs}(y(i)), \text{AbsTol}(i))$$

Per impostare valori differenti per questi parametri, bisogna quindi fare ricorso a **ODESET**.

ODESET consente di impostare DIVERSI **parametri** per la **risoluzione delle ODE** (per una trattazione più approfondita, si rimanda all'**help di Matlab**); nel nostro caso, abbiamo impostato solo i valori di **tolleranza relativa** ed **assoluta** da utilizzare con **ODE45** (bisogna passare le impostazioni definite con **odeset** alle **funzioni della libreria ODE di Matlab** come **quarto parametro**).

* * *

Extra - Tutorial 2

Tips: perché i e j non dovrebbero essere usati come nomi di variabili?

Quando si programma, l'utilizzo di *"i"* e *"j"* come variabili è piuttosto comune, soprattutto all'interno di **cicli for**, a volte anche annidati: in quei casi, nove volte su dieci si fa ricorso a *"i"* e *"j"* per il ciclo più esterno e per quello più interno, rispettivamente (e, a seguire, alle lettere *"h"* o *"k"*).

In **Matlab**, l'utilizzo di queste due lettere come nomi di variabili (ad es., in un ciclo: *"for i = 1:10"*) o indici di posizione (ad es., per una matrice: *"matrice(i, j) = x"*) è, invece, **fortemente sconsigliato**... perché?

Il motivo sta nel fatto che *"i"* e *"j"* sono, in effetti, funzioni native di **Matlab** che riguardano la definizione e l'utilizzo dei **numeri complessi** (e, nello specifico, dell'**unità immaginaria**), infatti scrivendo *"i"* o *"j"* nella **Command Window** e premendo Invio avremo, in entrambi i casi:

```
ans =  
0.0000 + 1.0000i
```

che, ovviamente, cambia se ad esempio utilizziamo *"i"* e *"j"* in altri contesti (in quanto sovrascriveremmo i valori originali).

... e in caso di errore, con una sovrascrittura accidentale di quei valori, perdiamo gli originali?

In caso di sovrascrittura di *"i"* e *"j"*, possiamo tornare ai loro valori originali con **clear**, scrivendo ad esempio:

```
clear i
```

e premendo Invio. In questo modo, cancelleremo dalla memoria di lavoro il nuovo valore dato a *"i"*, che riprenderà quello originale assegnatogli da **Matlab**.

... quindi non c'è soluzione? Dobbiamo solo stare attenti e utilizzare per forza altri indici o nomi di variabili?

In realtà, una soluzione c'è: lo stesso valore

```
ans =  
0.0000 + 1.0000i
```

è dato, in Matlab, da **"*i*"** e da **"*j*"**, ovvero digitando o utilizzando proprio:

```
1i
```

oppure

```
1j
```

nella Command Window o in uno script.

Si tratta di due scritture che **non possono essere... sovrascritte** (scusate il gioco di parole), perché **non possiamo** definire nomi di variabili che iniziano con dei numeri, infatti scrivendo ad esempio:

```
1i = 10
```

nella Command Window e premendo Invio, Matlab ci restituirà errore dicendo che il termine a sinistra dell'uguale non è un nome valido per un'assegnazione.

Quanto detto può esservi utile anche per interpretare script altrui, nel caso doveste trovare **"*i*"** e **"*j*"** da qualche parte in un codice... beh, adesso sapete a cosa si riferiscono questi particolari elementi!

In definitiva: se potete, evitate di usare **"*i*"** e **"*j*"**, perché potrebbero servire quando si ha a che fare con i numeri complessi; tuttavia, potete sempre ripulire i valori con **"clear *i*"** e **"clear *j*"** e, in ogni caso, se vi servono quei particolari valori potete utilizzare le parole-chiave **"1*i*"** e **"1*j*"**, native di **Matlab**.

```
*      *      *  
*      *      *
```