Università degli Studi di Verona

DIPARTIMENTO DI INFORMATICA
Laurea in Informatica

Riassunto di MATLAB

Basato sulle diapositive del corso

Autore:
Edoardo Righi

Anno Accademico 2017/2018

Indice

1 Introduzione a MATLAB

MATLAB sta per MATrix LABoratory. É un programma nato per gestire le matrici, ma ha molte applicazioni matematiche e grafiche. Nonostante abbia costrutti tipici della programmazione **NON** è un linguaggio di programmazione. Ha inoltre molte funzioni incorporate.

MATLAB può essere utilizzato con la sua *Command Window* (come un terminale) ma è anche possibile creare dei programmi.
Nella *Command Window* il '>>' indica il prompt, dove inserire il comando o l'espressione, a cui MATLAB risponderà con un risultato.

1.1 Ambiente Desktop

Le finestre dell'ambiente del programma sono:

- Cartella corrente: mostra il contenuto della cartella impostata come *Corrente*, dove i file verranno salvati;
- *Workspace*: mostra le variabili;
- Cronologia comandi: mostra gli ultimi comandi inseriti

La parte superiore dello schermo ha 3 schede: HOME (default), PLOTS e APPS. HOME è divisa in sezioni funzionali: FILE, VARIABLE, CODE, ENVIRONMENT, RESOURCES. Sotto ENVIRONMENT, con Layout è comunque possibile modificare l'ambiente Desktop.

1.2 Variabili

Per salvare un valore, si utilizza una variabile. Un modo per inserire un valore in una variabile è attraverso una *dichiarazione di assegnamento*.
Forma generale:

```
variabile = espressione
```

L'ordine è importante:

- Nome della **variabile** a sinistra;
- Operatore di assegnamento '=' (non rappresenta l'uguaglianza!);
- **espressione** a destra.

Ad esempio, inserendo nella *Command Window*:

```
>> mynum = 6
mynum =
     6
>>
```

Il risultato sarà l'assegnamento del risultato dell'espressione, 6, alla variabile *mynum*.
Un punto e virgola alla fine del comando sopprime l'output ma esegue comunque l'assegnamento:

```
>> mynum = 6;
>>
```

Infine, inserendo soltanto un espressione nel prompt, il risultato verrà salvato in una variabile di default chiamata *ans* che viene riutilizzata ogni volta che si inserisce un'espressione.

```
>> 7 + 4
ans =
    11
>>
```

1.2.1 Modifica di variabili

- Inizializzare una varibile (inserire il suo primo valore):

  ```
  >> mynum = 5;
  ```

- Incrementare una variabile di 1:

  ```
  >> mynum = mynum + 1;
  ```

- Decrementare una variabile di 2:

  ```
  >> mynum = mynum - 2;
  ```

1.2.2 Nomi delle variabili

I nomi di una variabile **devono** iniziare con una lettera dell'alfabeto. In seguito i nomi possono contenere lettere, numeri, e il carattere underscore '_'.
MATLAB è case-sensitive.
La funzione inclusa `namelengthmax` riporta il limite di lunghezza per il nome di una variabile. I comandi `who` e `whos` invece mostrano le variabili create (il secondo con maggior dettaglio), che possono poi essere eliminate con `clear`.

1.2.3 Tipi delle variabili

Ogni espressione e variabile è associata ad un *tipo*, o *classe*.

- Numeri reali: **single**, **double**
- Interi (i numeri nei nomi sono i numeri di bits usati per salvare il valore di quel tipo)
  - Signed: **int8**, **int16**, **int32**, **int64**
  - Unsigned: **uint8**, **uint16**, **uint32**, **uint64**
- Caratteri e stringhe: **char**
- True/False: **logical**

Il tipo di default è **double**.

1.3 Espressioni

Le espressioni possono contenere valori, variabili che sono già state create, operatori, funzioni integrate e parentesi.
Gli operatori includono:

+ Addizione

- Sottrazione

* Moltiplicazione

/ Divisione ($10/5 = 2$)

\ Divisione opposta ($5\backslash 10 = 2$)

^ Esponenziale ($5^2 = 25$)

La precedenza degli operatori è:

() Parentesi

^ Elevazioni a potenze

- Negazione

*,/,\ Moltiplicazione e division

+,- Addizione e sottrazione

1.4 Funzioni e Help

Esistono molte funzioni integrate in MATLAB. Funzioni correlate sono raggruppate in *help topics*.
Per vedere una lista degli *help topics*, basta scrivere "`help`" nel prompt:

```
>> help
```

Per trovare le funzioni in un *help topic*, ad esempio in elfun:

```
>> help elfun
```

Per invece cercare a proposito di una particolare funzione, ad esempio sin:

```
>> help sin
```

Per usare una funzione, basta *chiamarla*. Per chiamare una funzione, si scrive il suo nome seguito dagli argomenti che si vogliono passare dentro a delle parentesi. Molte funzioni calcolano valori e ritornano il risultato.
Ad esempio, per trovare il valore assoluto di -4:

```
>> abs(-4)
ans =
    4
```

Il nome della funzione è abs, a cui viene passato un argomento, -4. La funzione trova il valore assoluto e ritorna il risultato, 4.

Tutti gli operatori hanno inoltre una forma come funzione. Ad esempio un'espressione come 2+5 può essere scritta usando la funzione **plus** e passandovi 2 e 5 come argomenti:

```
>> plus(2,5)
ans =
    7
```

1.5 Costanti

Nella programmazione, le variabili sono usate per valori che possono cambiare, o che non sono conosciuti in anticipo.
Le *costanti* sono usate quando il valore è conosciuto e non può cambiare.

Esempi di costanti in MATLAB (sono effettivamente funzioni che ritornano valori costanti):

pi 3.14159...

i,j $\sqrt{-1}$

inf infinito

NaN significa "Not a Number" (risultato di 0/0)

1.6 Numeri random

Diverse funzioni integrate generano numeri random (pseudo-random in realtà). Funzioni di numeri random, o generatori di numeri random, iniziano con un numero chiamato seme (**seed**); questo è o un valore predeterminato o dipendente dall'ora corrente. Di default MATLAB usa un valore predeterminato, quindi sarà sempre lo stesso.
Per impostare il seme usando l'ora:

```
rng('shuffle')
```

1.6.1 Numeri random reali

La funzione **rand** genera numeri reali uniformamente distribuiti nell'intervallo aperto (0,1). Chiamandola senza argomenti ritorna un numero random reale.
Per generare un numero reale nell'intervallo aperto (0,N) basta moltiplicare questa funzione per N:

```
rand * N
```

randn viene usato per generare una distribuzione normale di numeri random reali.

1.6.2 Numeri random interi

Arrotondare un numero random reale potrebbe essere usato per produrre un intero, ma questi interi non verrebbero distribuiti uniformemente nell'intervallo.

La funzione **randi(imax)** genera un intero random nell'intervallo tra 1 e imax, incluso.
É anche possibile passare un intervallo:

```
randi([m,n],1)
```

che genera un intero nell'intervallo tra m e n.

1.7 Caratteri e stringhe

Un **character** è un singolo carattere tra apici. Tutti i caratteri nel set di caratteri di un computer sono ordinati usando una **codifica di caratteri**. I set di caratteri includono tutte le lettere dell'alfabeto, i numeri, i segni di punteggiatura, gli spazi, ecc.

Le stringhe di caratteri sono sequenze di caratteri tra doppie virgolette (ad esempio, "ciao, come stai?"). Nella sequenza di codifica di caratteri, le lettere dell'alfabeto sono in ordine (ad esempio, 'a' viene prima di 'b').

La comune codifica ASCII ha 128 caratteri (0-127 gli equivalenti interi), ma MATLAB supporta sequenze di codifica molto più grandi.
Il range dei tipi interi può essere trovata con **intmin**/**intmax** (ad esempio, intmin('int8') è -128, intmax('int128') è 127).

Convertire da un tipo all'altro, usando qualsiasi nome di tipo come funzione, è chiamato *casting* o *casting di tipo*. Ad esempio:

```
>> num = 6 + 3;
>> numi = int32(num);
>> whos
   Name      Size     Bytes   Class    Attributes

   num       1x1      8       double
   numi      1x1      4       int32
```

La funzione **class** ritorna il tipo di una variabile.
La codifica ASCII standard ha 128 caratteri, i cui equivalenti interi sono 0-127.
Ogni funzione di numero può convertire un carattere nell'equivalente intero:

```
>> numequiv = double('a')
numequiv =
    97
```

La funzione **char** invece converte un intero nel carattere equivalente.

1.8 Espressioni relazionali

Gli operatori relazionali in MATLAB sono:

- $>$ maggiore di
- $<$ minore di

>= maggiore o uguale

<= minore o uguale

== uguaglianza

~= disuguaglianza

Il tipo del risultato è **logical**: 1 per Vero o 0 per falso.
Gli operatori logici sono:

|| OR per gli scalari

&& AND per gli scalari

~ NOT

In più la funzione **XOR**, che ritorna Vero logico solamente se uno solo degli argomenti è Vero.

Operatori	Precedenza
parentesi: ( )	maggiore
potenza: ^	
unari: negazione (-) e not (~)	
moltiplicazione, divisione: *, /, \	
addizione, sottrazione: +, -	
relazioniali: <, <=, >, >=, ==, ~=	
AND: &&	
OR: ||	
assegnamento: =	minore

Tabella 1: Tabella delle precedenze espansa

1.9 Altre funzioni

Altre funzioni di MATLAB sono:

- Le funzioni trigonometriche come **sin**, **cos**, **tan** (in radianti).
  - Anche arcoseno **asin** , sendo iperbolico **hsin**, ecc.
  - Funzioni che usano i gradi: **sind**, **cosd**, **asind**, ecc.
- Funzioni di arrotondamento e resto:
  - **fix**, **floor**, **ceil**, **round**
  - **rem**, **mod** ritornano il resto
  - sign ritorna il segno come -1, 0 o 1
- Funzioni **sqrt** e **nthroot**.
- **deg2rad** e **rad2deg** convertono tra radianti e gradi, o viceversa.

- Funzioni logaritmiche:
  - **log(x)** ritorna l'algoritmo naturale (base e)
  - **log2(x)** ritorna l'algoritmo di base 2
  - **log10(x)** ritorna l'algoritmo di base 10
- Funzione esponenziale exp(n) che ritorna la costante e^n.
  - Nota: non esiste nessuna costante integrata per rappresentare e; si usa **exp** per ottenerla.
  - Non confondere con la notazione esponenziale e.

2 Matrici

Una **matrice** viene usata per salvare un set di valori dello stesso tipo; ogni valore è salvato in un **elemento** della matrice.
Una matrice assomiglia ad una tabella, con righe e colonne; una matrice di m righe e n colonne viene chiamata $m\ x\ n$; questi due valori sono chiamati le **dimensioni** della matrice. Ad esempio una matrice 2x3 è:

9	6	3
5	7	2

Il termine **array** viene frequentemente usato in MATLAB per riferirsi genericamente ad una matrice o ad un vettore.

2.1 Vettori e Scalari

Un **vettore** è un caso speciale di matrice dove una delle dimensioni è 1.

- Un vettore con n elementi è 1xn, ad esempio 1x4:

5	88	3	11

- Un vettore con m elementi è mx1, ad esempio 3x1:

3
7
4

Uno **scalare** è un caso ancora più speciale: corrisponde ad una matrice 1x1, cioè ad un singolo valore.

2.1.1 Creare righe di vettori

- Metodo diretto: mettere i valori desiderati tra parentesi quadrate, separati o da spazi o da virgole

```
>> v = [1 2 3 4]
v =
     1 2 3 4
>> v = [1,2,3,4]
v =
     1 2 3 4
```

- Operatore "**due punti**": itera attraverso i valori con la forma *first:step:last*. Ad esempio 5:3:14 ritorna il vettore [5 8 11 14]
  - Se non specificato, *step* vale 1
  - Si può andare al contrario, ad esempio 4:-1:1 crea [4 3 2 1]

2.1.2 Funzioni linspace e logspace

La funzione **linspace** crea un vettore "linearmente spaziato"; **linspace(x,y,n)** crea un vettore con n valori nel range inclusivo da x a y.
Ad esempio linspace(4,7,3) crea un vettore con 3 valori, inclusi il 4 e il 7, cioè il vettore: [4 5.5 7].
Se n non è definito, vale 100 punti.

La funzione **logspace** crea invece un vettore "logaritmicamente spaziato"; **logspace(x,y,n)** crea un vettore con n valori nel range inclusivo tra 10^x e 10^y.
Ad esempio logspace(2,4,3) ritorna [100 1000 10000].
Se n non è definito vale 50 punti.

2.1.3 Concatenazione

I vettori possono essere creati dall'unione di vettori esistenti, o aggiungendo valori a vettori esistenti: questo viene detto **concatenazione**.
Per esempio:

```
>> v = 2:5;
>> x = [33 11 2];
>> w = [v x]
w =
     2 3 4 5 33 11
>> newv = [v 44]
newv =
     2 3 4 5 44
```

2.1.4 Riferirsi ad elementi

Gli elementi in un vettore sono numerati sequenzialmente; ogni numero di elemento è detto **indice**. Sono indicati sopra gli elementi nell'esempio di seguito:

1	2	3	4	5
5	33	11	-4	2

Si può fare riferimento ad un elemento usando il suo indice tra parentesi; ad esempio vec(4) è il quarto elemento del vettore *vec*.
Si può inoltre fare riferimento ad una parte di un vettore usando un *vettore indice*: vec([2 5]) si riferisce al secondo e al quinto elemento di vec; vec([1:4]) si riferisce ai primi 4 elementi del vettore.

2.1.5 Modifica di vettori

Elementi di un vettore possono essere cambiati; ad esempio con

$$vec(3) = 11$$

il terzo elemento di vec viene sostituito con 11. Un vettore può essere esteso riferendosi ad elementi che non esistono ancora; se c'è un vuoto tra la fine del vettore e il nuovo elemento specificato(/i nuovi elementi specificati), viene colmato da zeri. Ad esempio:

```
>> vec = [3 9];
>> vec(4:6) = [33 2 7]
vec =
    3 9 0 33 2 7
```

L'estensione dei vettori non è una buona idea se può essere evitata comunque.

2.1.6 Creare colonne di vettori

Un vettore colonna è un vettore mx1.

- Metodo diretto: mettere i valori desiderati tra parentesi quadrate, separati da un punto e virgola, ad esempio [4;7;2]
- Non è possibile creare direttamente un vettore colonna usando metodi come l'operatore "due punti", ma si può creare utilizzando questo operatore un vettore riga e poi farne la **trasposizione** (trasposta di A con A').

Il riferimento agli elementi funziona allo stesso modo dei vettori riga.

2.2 Creare variabili matrici

Per creare matrici si separano i valori tra righe con spazi o virgole, e si separano le diverse righe con un punto e virgola. Si può utilizzare qualsiasi metodo per ottenere i valori in ogni riga (cioè ad ottenere un vettore riga, quindi anche l'operatore "doppi punti").

```
>> mat = [1:3; 6 11 -2]
mat =
    1  2  3
    6 11 -2
```

Devono sempre esserci gli stessi numeri di valori in ogni riga!

2.2.1 Funzioni che creano matrici

Esistono molte funzioni integrate per creare matrici:

- **rand(n)** crea una matrice nxn di reali random
- **rand(n,m)** crea una matrice nxm di reali random
- **randi([range],n,m)** crea una matrice nxm di interi random nell'intervallo *range* specificato
- **zeros(n)** crea una matrice nxn di zeri
- **zeros(n,m)** crea una matrice nxm di zeri
- **ones(n)** crea una matrice nxn di uni
- **ones(n,m)** crea una matrice nxm di uni

2.2.2 Elementi di una matrice

Per riferirsi ad un elemento all'interno di una matrice, si usa il nome della matrice seguito da indice della riga e della colonna tra parentesi, divisi da una virgola:

```
>> mat = [1:3; 6 11 -2]
mat =
     1   2   3
     6  11  -2
>> mat(2,1)
ans =
     6
```

Riferirsi sempre prima alla riga e poi alla colonna!

Ci si può inoltre riferire ad un sottoinsieme di una matrice:

- Per riferirsi all'intera m-esima riga: **mat(m,:)**
- Per riferirsi all'intera n-esima colonna: **mat(:,n)**

Inoltre per riferirsi alle ultime righe o colonne si può usare **end**; ad esempio mat(end,m) è la colonna m-esima dell'ultima riga.

2.2.3 Modifica di matrici

Un elemento individuale in una matrice può essere modificato assegnandoli un nuovo valore; anche intere righe e colonne possono essere modificate allo stesso modo.
Ogni sottoinsieme di una matrice può essere modificato, finchè il nuovo sottoinsieme assegnato abbia la stessa dimensione dell'originale.
Un'eccezione: uno scalare può essere assegnato ad un sottoinsieme di qualsiasi dimensione: lo stesso scalare viene assegnato ad ogni elemento nel sottoinsieme.

2.2.4 Dimensione delle matrici

Esistono diverse funzioni per determinare la dimensione di vettori o matrici:

- **length(vec)** ritorna il numero di elementi in un vettore
- **length(mat)** ritorna la dimensione maggiore (tra righe e colonne) di una matrice
- **size** ritorna il numero di righe e colonne di un vettore o una matrice.
  Nel caso di una matrice, bisogna catturare entrambi i risultati:

  ```
  [r c] = size(mat)
  ```

- **numel** ritorna il numero totale di elementi in un vettore o una matrice

Rimanere generici nella programmazione è molto importante: invece che assumere di conoscere la dimensione di vettori o matrici, meglio usare **length** o **size** per scoprirlo.

Molte funzioni permettono di modificare la dimensione delle matrici:

- **reshape** cambia le dimensioni di una matrice in una con lo stesso numero di elementi
- **rot90** ruota una matrice di 90 gradi (senso anti-orario)
- **fliplr** ribalta le colonne di una matrice da sinistra a destra
- **flipud** ribalta le righe di una matrice dall'alto verso il basso
- **flip** ribalta un vettore riga da sinistra a destra, vettore colonna o matrice dall'alto verso il basso

2.2.5 Replicare matrici

- **repmat** replica un'intera matrice; crea $m\ x\ n$ copie della matrice
- **repelem** replica ogni elemento di una matrice nelle dimensioni specificate

```
>> mymat = [33 11; 4 2]
mymat =
    33 11
     4  2
>> repmat(mymat, 2,3)
ans =
    33 11 33 11 33 11
     4  2  4  2  4  2
    33 11 33 11 33 11
     4  2  4  2  4  2
>> repelem(mymat,2,3)
ans =
    33 33 33 11 11 11
    33 33 33 11 11 11
     4  4  4  2  2  2
     4  4  4  2  2  2
```

2.3 Vettori vuoti

Un vettore vuoto è un vettore senza elementi; un vettore vuoto può essere creato usando le parentesi quadre con niente all'interno.
Per eliminare un elemento da un vettore, basta assegnare un vettore vuoto a quell'elemento. Nel caso di una matrice non è possibile eliminare un singolo elemento, ma si può eliminare una riga o una colonna intera assegnandovi [ ].
Vettori vuoti possono anche essere utilizzati per "accrescere" un vettore, iniziando con nulla e aggiungendogli valori attraverso concatenazione (di solito in un loop, che verrà trattato più avanti).
Questo tuttavia non è efficiente, e andrebbe evitato quando possibile.

2.4 Matrici 3D

Una matrice tridimensionale ha grandezza $m\ x\ n\ x\ p$.
Possono essere create utilizzando funzioni integrate; ad esempio di seguito viene creata una matrice 3x5x2 di interi random: ci sono 2 strati, ognuno di dimensioni 3x5

```
>> randi([0 50], 3,5,2)
ans(:,:,1) =
    36  34   6  17  38
    38  33  25  29  13
    14   8  48  11  25

ans(:,:,2) =
    35  27  13  41  17
    45   7  42  12  10
    48   7  12  47  12
```

2.5 Array come argomenti di funzioni

Interi array (vettori o matrici) possono essere passati come argomenti di funzioni. Il risultato avrà la stessa dimensione dell'input.
Ad esempio:

```
>> vec = randi([-5 5], 1, 4)
vec =
    -3 0 5 1
>> av = abs(vec)
av =
    3 0 5 1
```

2.6 Funzioni su array

Esistono numerose funzioni integrate molto utili per operare su vettori, o su colonne di matrici:

- **min** ritorna il minimo valore
- **max** ritorna il massimo valore
- **sum** ritorna la somma degli elementi
- **prod** ritorna il prodotto degli elementi
- **cumprod** prodotto cumulativo
- **cumsum** somma cumulativa
- **cummin** minimo cumulativo
- **cummax** massimo cumulativo

2.6.1 Esempi con min, max

```
>> vec = [4 -2 5 11];
>> min(vec)
ans =
    -2
>> mat = randi([1, 10], 2,4)
```

```
mat =
    6   5   7   4
    3   7   4  10
>> max(mat)
ans =
    6   7   7  10
```

2.6.2 Esempi con sum, cumsum

La funzione **sum** ritorna la somma degli elementi; la funzione **cumsum** mostra la somma mentre itera attraverso gli elementi (4, poi $4 + -2$, poi $4 - 2 + 5$ e infine $4 - 2 + 5 + 11$):

```
>> vec = [4 -2 5 11];
>> sum(vec)
ans =
    18
>> cumsum(vec)
ans =
    4   2   7  18
```

2.6.3 Esempi con prod, cumprod

Queste funzioni hanno lo stesso formato di **sum**/**cumsum**, ma calcolano i prodotti:

```
>> v = [2:4 10]
v =
    2   3   4  10
>> cumprod(v)
ans =
    2   6  24 240
>> mat = randi([1, 10], 2,4)
mat =
    2   2   5   8
    8   7   8  10
>> prod(mat)
ans =
    16  14  40  80
```

2.7 Funzioni generali su matrici

Visto che queste funzioni operano secondo colonne per matrici, è necessario annidare per colonne le chiamate, in modo da ottenere la funzione per tutti gli elementi di una matrice:

```
>> mat = randi([1, 10], 2,4)
mat =
    9   7   1   6
    4   2   8   5
>> min(mat)
ans =
```

```
    4   2   1   5
>> min(min(mat))
ans =
    1
```

2.8 Funzione diff

La funzione **diff** ritorna la differenza tra elementi consecutivi in un vettore. Per un vettore di lunghezza *n*, la lunghezzza del risultato sarà *n-1*:

```
>> diff([4 7 2 32])
ans =
    3 -5 30
```

Nel caso di una matrice, la funzione diff trova le differenze per colonne.

2.9 Operazioni scalari

Operazioni numeriche possono essere eseguite su ogni elemento in un vettore o una matrice. Per esempio, le **moltiplicazioni scalari**: moltiplicare ogni elemento per uno scalare:

```
>> [4 0 11] * 3
ans =
    12   0 33
```

Un altro esempio: addizioni scalari; aggiungere uno scalare ad ogni elemento:

```
>> zeros(1,3) + 5
ans =
    5   5   5
```

2.10 Operazioni su array

Su due matrici A e B vengono applicate termine per termine, o elemento per elemento: questo significa che le matrici devono avere le stesse dimensioni.

- addizione di matrici: $A + B$
- sottrazione di matrici: $A - B$ o $B - A$

Per operazioni basate sulla moltiplicazione (moltiplicazione, divisione, esponenziale), è necessario posizionare un operatore davanti all'operatore.

- moltiplicazione di array: $A.^*B$
- divisione di array: $A./B$ o $A. \setminus B$
- esponziale di array: $A.\hat{}2$

La moltiplicazione di matrici NON è un operazione su array.

2.11 Vettori logici

Usare operatori relazionali su un vettore o una matrice risulta in un vettore o una matrice logici.

```
>> vec = [44 3 2 9 11 6];
>> logv = vec > 6
logv =
    1   0   0   1   1   0
```

Si può usare questo per indicizzare un vettore o una matrice (solo se il vettore indice è di tipo logico):

```
>> vec(logv)
ans =
    44   9 11
```

2.11.1 True/False

Gli operatori logici sono:

- **false** equivale a logical(0)
- **true** equivale a logical(1)

false e **true** sono anche funzioni che creano matrici di valori false o true.
Da R2016a questo può essere fatto anche con **ones** e **zeros**.

2.11.2 Funzioni logiche

- **any** ritorna true se almeno un argomento in input è vero
- **all** ritorna true solo se l'intero argomento in input è vero
- **find** trova le locazioni e ritorna gli indici

```
>> vec
vec =
    44   3   2   9 11   6
>> find(vec>6)
ans =
     1   4   5
```

2.11.3 Comparazione di array

La funzione **isequal** compara due array, e ritorna **true** se sono uguali (cioè se tutti gli elementi corrispondono) o **false** altrimenti.

```
>> v1 = 1:4;
>> v2 = [1 0 3 4];
>> isequal(v1,v2)
ans =
    0
```

```
>> v1 == v2
ans =
     1   0   1   1
>> all(v1 == v2)
ans =
     0
```

2.11.4 Operatori su elementi

- | e & sono usati per matrici: passano elemento per elemento e ritornano i valori logici 1 o 0
- || e && sono usati per gli scalari

2.12 Moltiplicazione matriciale

La moltiplicazione matriciale NON è un operazione per array: non significa moltiplicare termine per termine.
In MATLAB, l'operatore di moltiplicazione * esegue una moltiplicazione matriciale. Affinchè sia possibile moltiplicare una matrice A per una matrice B è necessario che il numero di colonne di A sia lo stesso del numero di righe di B.

Ad esempio se A ha dimensioni mxn, significa che B dovrà avere dimensioni nxp: la dimensione interiore deve essere la stessa. La matrice risultante C ha lo stesso numero di righe di A e di colonne di B (mxp).
Gli elementi della matrice C sono calcolati dalla somma dei prodotti di elementi corrispondenti nelle righe di A e nelle colonne di B:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

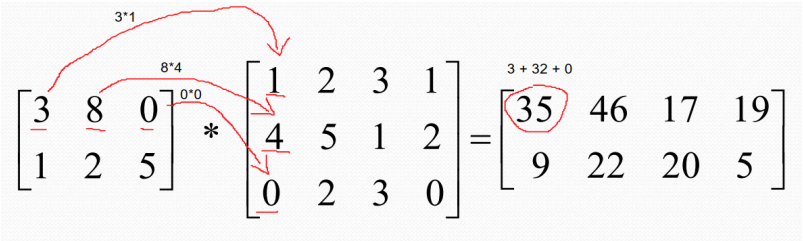

Figura 1: Esempio di moltiplicazione matriciale.

2.13 Operazioni su vettori

Dato che i vettori sono casi speciali di matrici, le operazioni su matrici descritte, incluse addizione, sottrazione, moltiplicazione scalare, moltiplicazione e trasposta, funzionano anche su vettori, se le dimensioni sono corrette.
Operazioni specifiche su vettori sono:

- Il **prodotto scalare** o prodotto interiore di due vettori a e b è definito come $a_1b_1 + a_2b_2 + ... + a_nb_n$.
  - La funzione integrata **dot** esegue questa operazione
- Inoltre, **cross** esegue il prodotto vettoriale

3 Script e funzioni

Un algoritmo è una sequenza di passi necessaria a risolvere un problema.
Un approcio alla programmazione di tipo **top-down** consiste nel frammentare la soluzione in passi, per poi rifinirne ognuno.
Un algoritmo generico per molti programmi è:

1. Prendere l'input
2. Calcolare il risultato (o i risultati)
3. Mostrare il risultato (o i risultati)

Un programma modulare consiste in funzioni che implementano ogni passo.

Gli script sono file in MATLAB che contengono una sequenza di istruzioni MATLAB, implementando un algoritmo. Gli script vengono interpretati, e sono salvati in file di codice (con estensione *.m*).
Nel momento in cui uno script viene salvato, è possibile eseguirlo inserendo il suo nome nel prompt. Il comando **type** può essere utilizzato per mostrare uno script nella finestra dei comandi. Uno script dovrebbe sempre essere **documentato** usando **commenti**. I commenti descrivono cosa fa uno script e in quale modo; vengono ignorati da MATLAB. Per commentare si può:

- Aggiungere il simbolo *%* prima dell'inizio di un commento, che finisce al termine della riga
- Inserire il commento tra *%{* e *%}* per commenti di più righe

In particolare, la prima linea di commento in uno script è detta 'H1 line' ed è quello che viene mostrato con il comando **help**.

3.1 Input e Output

La funzione **input** fa due cose: richiede all'utente un valore e lo legge.

- Forma generale di lettura di un numero:

```
variablename = input('prompt string')
```

- Forma generale di lettura di una stringa o un carattere:

```
variablename = input('prompt string', 's')
```

Per leggere più valori sono necessari altrettante funzioni input.

Per l'output esistono due funzioni base:

- **disp**, che è un modo veloce per visualizzare cose
- **fprintf**, che permette la formattazione

La funzione **fprintf** usa formattazione delle stringhe che include placeholders; questi hanno caratteri di conversione:

%d interi

%f float (numeri reali)

%c caratteri singoli

%s stringhe

Usando *%#x*, dove # è un intero e x il carattere di conversione, è possibile specificare la larghezza del campo pari a #.
Con *%#.#x* è invece possibile specificare la larghezza del campo e il numero di valori decimali.
%.#x specifica solo il numero dei decimali (o caratteri in una stringa); la larghezza di campo verrà invece espansa di quanto necessaria.

Altra formattazione:

- \n è il carattere per passare ad una nuova linea
- \t è il carattere di tabulazione
- la giustificazione a sinistra è possibile con '-' (esempio: %-5d)
- per stampare uno slash: \\
- per stampare un singolo apice: '' (due virgolette singole)

La stampa di vettori e matrici è solitamente più facile con **disp**.

3.2 Grafici semplici

Usando **plot** è possibile creare semplici grafici di punti di dati.
Per iniziare, si creano variabili per salvare i dati (possono contenere uno o più punti ma devono essere della stessa lunghezza); ad esempio presi due vettori chiamati x e y (o senza x se i suoi valori sono 1,2,3,ecc):

```
plot(x,y)    o solo   plot(y)
```

Di default i punti individuali sono tracciati con segmenti che li uniscono, ma altre opzioni possono essere specificate con un argomento aggiuntivo che è una stringa. Le opzioni possono includere colori (ad esempio 'b' per blu, 'g' per verde, ecc), simboli di grafico o marcatori (ad esempio 'o' per cerchi, '+', '*'), tipi di linea (ad esempio '–' per la tratteggiata).

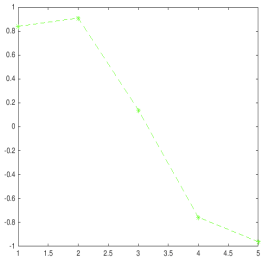

Figura 2: Esempio di plot(x,y, 'g*–') con x=[1,2,3,4,5] e y=sin(x)

Di default non ci sono labels sugli assi o titoli nel grafico. Per aggiungerli si possono usare le funzioni:

- xlabel('stringa')
- ylabel('stringa')
- title('stringa')

Gli assi sono creati di default usando il minimo e il massimo valore nei vettori dati x e y. Per specificare range differenti si usa la funzione **axis**:

- axis([xmin xmax ymin ymax])

Altre funzioni utili per un grafico:

- **clf**, per pulire la finestra della figura
- **figure**, crea una nuova finestra figura
- **hold**, mantiene il grafico corrente nella finestra figura
- **legend**, mostra la legenda
- **grid**, mostra le linee della griglia
- **bar**, crea un grafico a barre

3.3 File I/O

Esistono 3 modalità o operazioni su file:

- read from
- write to (assumendo che cominci dall'inizio)
- append to (come write to, ma iniziando dalla fine)

Esistono semplice comandi per salvare una matrice in un file e leggere da un file in una matrice: **save** e **load**. Per leggere o scrivere qualcosa di diverso da una matrice, sono necessarie funzioni I/O di più basso livello.

3.3.1 load e save

Per leggere da un file in una variabile matrice:

```
load filename.ext
```

Questo creerà una variabile matrice chiamata "filename". Può essere usata se il file ha lo stesso numero di valori su ogni linea nel file; ogni linea è letta in una riga nella matrice variabile.
Per scrivere il contenuto di una variabile matrice in un file:

```
save filename matrixvariablename -ascii
```

Per appendere il contenuto di una variabile matrice in un file esistente:

```
save filename matrixvariablename -ascii -append
```

3.4 Funzioni definite dall'utente

Esistono diversi tipi di funzioni che possono essere create dall'utente. Nel caso di una funzione che calcola e ritorna un valore, si scrive la definizione della funzione (salvata in un codice con estensione *.m*) e, allo stesso modo delle funzioni incluse, la si chiama passando uno o più argomenti tra parentesi, che la funzione elaborerà per ottenere il risultato, che viene poi ritornato.

```
function outarg = fnname(argomenti input)
    del codice qui
    outarg = un valore;
end
```

La definizione include:

- la funzione header (prima linea)
- la funzione body (tutto il resto)

3.4.1 Header

L'header inizia sempre con la parola riservata *function*, poi c'è il nome di un argomento di output, seguito dall'operatore assegnamento e il nome della funzione (dovrebbe essere lo stesso del nome del file) con gli argomenti tra parentesi (corrispondenti uno a uno con gli argomenti che si passano con la chiamata).

3.4.2 Esempio funzione

```
function area = calcarea(rad)
    % Questa funzione calcola l'area di un cerchio
    area = pi * rad * rad;
end
```

Questa funzione potrebbe essere chiamata in diversi modi:

- »calcarea(4)
  Questo salva il risultato nella variabile di default *ans*
- »myarea = calcarea(4)
  Questo salva il risultato nella variabile myarea
- »disp(calcarea(4))
  Questo mostra il risultato, ma non lo salva per uso successivo

N.B.: questa funzione non accetta arrey a causa dell'operatore prodotto, per permetterlo basta modificarlo in '.*'.

4 Simulink

Booooooooooooo

5 Istruzioni di selezione

5.1 IF

L'istruzione **if** viene usata per determinare se un istruzione o un gruppo di istruzioni deve essere eseguito:

```
if condizione
    azione
end
```

5.2 IF-ELSEIF-ELSE

Le istruzioni **if-elseif-else** permettono di decidere tra due o più azioni:

```
if condizione1
    azione1
elseif condizione2
    azione2
else
    azione3
end
```

5.3 SWITCH

L'istruzione **switch** può essere usata frequentemente al posto di una serie di if annidati:

```
switch espressione_switch
    case caso1
        azione1
    case caso2
        azione2
    case caso3
        azione3
    otherwise
        azione4

end
```

5.4 funzioni IS

Esistono diverse funzioni "is" in MATLAB che essenzialmente chiedono se una cosa è vera/falsa, e ritornano il logico 1 (true) o 0 (false):

- **isletter** ritorna 1 o 0 se il carattere della stringa è una lettera o meno
- **isempty** ritorna 1 se l'argomento è una variabile vuota
- **iskeyword** ritorna 1 se la stringa argomento è una keyword
- **isa_** determina se l'argomento è uno specificato tipo (_)

6 Istruzioni di loop

6.1 FOR loop

Usato come loop contato, ripete un'azione uno specificato numero di volte. Un iteratore specifica quante volte ripetere l'azione:

```
for loopvar = range
    Azione
end
```

Il range viene specificato da un vettore (numInizio:numFine).
Altre funzioni che operano con vettori sono: **prod**, **cumsum**, **cumprod**, **min**, **max**,**cummin**, **cummax**.

6.2 subplot

La funzione **subplot** crea una matrice (o vettore) in una finestra Figure così da poter mostrare più grafici contemporaneamente. Se la matrice è mxn, la chiamata della funzione **subplot(m,n,i)** si riferisce all'elemento i (che deve essere un intero nel range da 1 a m*n). A volte è possibile usare un for loop per iterare tra i diversi valori di subplot.

6.3 WHILE loop

Usato come loop condizionale, ripete un'azione finchè la condizione non diventa falsa:

```
while condition
    action
end
```

7 Programmi MATLAB

Categorie di funzioni:

- funzioni che calcolano e ritornano un valore
- funzioni che calcolano e ritornano più di un valore
- funzioni che compiono semplicemente un compito, senza ritornare valori

Sono diversi nel:

- modo in cui sono chiamate
- aspetto dell'header

Sono tutte salvate in file con estensione '.m'.

7.1 Funzioni che ritornano >1 valori

Forma generale: hanno molteplici argomenti di output nell'header. Questi argomenti sono separati da virgole.
Per salvare tutti i valori ritornati, la chiamata dovrebbe essere assegnata ad un vettore contenente lo stesso numero di valori. Altrimenti alcuni di essi verranno persi.
Ad esempio una funzione con header:

```
function [x,y,z] = fnname(a,b)
```

ritorna 3 valori, quindi una chiamata a questa funzione dovrebbe essere qualcosa come:

```
[g,h,t] = fnname(11, 4.3);
```

Si possono anche usare gli stessi nomi nell'header. Una chiamata a funzione come la seguente salverebbe solo il primo valore ritornato:

```
result = fnname(11, 4.3);
```

7.2 Funzioni che non ritornano valori

Una funzione che non ritorna valori non ha valori di output nell'header, nè l'operatore di assegnamento:

```
function functionname(input argomenti)
    azioni
end
```

Allo stesso modo una chiamata a questa funzione è un'istruzione. Ad esempio con la funzione esempio precedente, una chiamata sarebbe:

```
functionname(x,y)
```

La presenza di un assegnamento invaliderebbe la chiamata!

7.3 Sottofunzioni

Quando una funzione chiama un'altra funzion, le due possono essere salvate nello stesso file con il nome della funzione principale. La sottofunzione potrà essere chiamata solo dalla funzione principale.

7.4 Tipi di errore

- *Errori di sintassi*: errori nel linguaggio
- *Run-time* (a tempo di esecuzione): errori trovati durante l'esecuzione di uno script o funzione
- *Errori logici*: errori nel ragionamento sul funzionamento

7.4.1 Metodi di debug

Esistono diversi metodi oer trovare gli errori:

- Tracing: usare **echo** per vedere come sono eseguite le istruzioni
- Usando il Debugger/Editor di MATLB
- Posizionando breakpoint per poter esaminare i valori di variabili ed espressioni in vari punti
  - **dbstop** per posizionare un breakpoint
  - **dbcont** per continuare l'esecuzione
  - **dbquit** per uscire dalla modalità debug

7.5 Code Cells

Il codice negli script può essere separato in sezioni chiamate **code cells**. È possibile eseguire un code cell alla volta. Queste celle vengono create con un commento che inizia con due percentuali '%%'.

8 Strutture dati

8.1 Cell arrays

Un **cell array** è un tipo di struttura dati che può immagazzinare diversi tipi di valori nei suoi elementi. Un cell array può essere un vettore (riga o colonna) o una matrice. Essendo un array, utilizza indici per riferirsi ai propri elementi. Un'ottima applicazione dei cell arrays: salvare stringhe di diversa lunghezza.

8.1.1 Creare Cell arryas

La sintassi per creare un cell array è con le parentesi graffe invece delle quadre. Il metodo diretto è di aggiungere valori nella riga (o nelle righe) separati da virgole o spazi, e di separare le righe con punti e virgola (come negli array).
La funzione **cell** può essere usata per preallocare passando la dimensione del cell array:

```
cell(4,2)
```

8.1.2 Riferirsi agli elementi

Gli elementi in un cell array sono celle.
Esistono due metodi di riferirsi alle parti di un cell array:

- riferirsi alla cella (**cell indexing**); si usano le parentesi tonde
- riferirsi al contenuto della cella (**content indexing**); si usano le graffe

```
>> ca = {2:4, 'hello'};

>> ca(1)     %cell indexing
ans =
    [1x3 double]

>> ca{1}     %content indexing
ans =
    2 3 4
```

8.1.3 Funzioni Cell arrays

- **celldisp**: mostra il contenuto di tutti gli elementi del cell array
- **cellplot**: apre un grafico delle celle (non del contenuto!)
- **cellstr**: converte da una matrice di caratteri ad un cell array di stringhe
- **iscellstr**: ritorna 1 se un cell array contiene solo stringhe
- **strjoin**: concatena tutte le stringhe in un cell array in una stringa, separate da un delimitatore (spazio di default ma altri possono essere specificati)
- **strspit**: split di una stringa negli elementi di in un cell array usando uno spazio come delimitatore di default (altri specificabili)

8.2 Strutture

Le strutture salvano valori di tipo diverso in **fields** (campi). I campi sono caratterizzati da nomi; ci si può riferire a loro attraverso **l'operatore punto** (structurename.fieldname). Le strutture possono essere inizializzate con la funzione **struct**, che prende coppie di argomenti (nome del campo come stringa, seguito dal valore di quel campo).
Per stampare, **disp** mostra tutti i campi, mentre **fprintf** può stampare solo i campi individuali.

```
>> subjvar = struct('SubjNo',123,'Height',62.5);
>> subjvar.Height
ans =
    62.5000
>> disp(subjvar)
    SubjNo: 123
    Height: 62.5000
>> fprintf('The subject # is %d\n', subjvar.SubjNo)
The subject # is 123
```

8.3 Funzioni Strutture

- **rmfield**: rimuove un campo ma non altera la variabile
- **isstruct**: ritorna 1 se l'argomento è una variabile struttura
- **isfield**: riceve una variabile struttura e una stringa, ritorna 1 se la stringa è il nome di un campo della struttura
- **fieldnames**: riceve una variabile struttura e ritorna i nomi di tutti i suoi campi come cell array

8.4 Cell arrays vs Strutture

- I Cell arrays sono array, quindi sono indicizzati (e quindi è possibile iterare sui loro elementi con un ciclo
- Le strutture non son indicizzate, tuttavia i nomi dei campi sono mnemonici quindi è più chiaro cosa si sta salvando in una struttura

8.5 Vettore di strutture

Un database di informazioni può essere salvato in MATLAB in un vettore di strutture, cioè un vettore in cui ogni elemento è una struttura.
Un esempio con dati medici:

```
>> subjvar(2) = struct('SubjNo', 123, 'Height',
    62.5, 'Weight', 133.3);
>> subjvar(1) = struct('SubjNo', 345, 'Height',
    77.7, 'Weight', 202.5);
```

In questo caso si crea un vettore con 2 strutture. La seconda struttura viene creata per prima così da preallocare per 2 elementi.
Un set di campi può essere creato, ad esempio:

```
>> [subjvar.Weight]
ans =
    202.5000  133.3000
```

8.6 Strutture annidate

Una struttura annidata è una struttura nella quale almeno un campo è un'altra struttura. Per riferirsi alla struttura "interna", l'operatore punto deve essere usato due volte (es.: structurename.innerstruct.fieldname). Per creare una struttura annidata basta annidare le chiamate alla funzione **struct**.
Esempio con nome e cognomi salvati separati in una struttura separata:

```
>> contactinfo = struct('cname',
        struct('last', 'Smith', 'first', 'Abe'),
        'phoneExt', '3456');
>> contactinfo.cname.last
ans =
    Smith
```

8.7 Tabelle

È possibile salvare informazioni nel formato di una tabella con righe e colonne, entrambe mnemonicamente etichettate. Le tabelle si possono creare con la funzione **table** che specifica variabili (le colonne) e nomi delle righe (cell array di stringhe). L'indicizzazione nella tabella può essere eseguita con indici interi o utilizzando le stringhe che sono la riga o i nomi della variabile. La funzione **summary** mostra i dati statistici (min, mediana, max) per ogni variabile.

8.8 Ordinamento

L'ordinamento può essere ascendente (dal minore al maggiore) o discendente (dal maggiore al minore). MATLAB ha incluse funzioni per ordinare. Esistono diversi algoritmi, come il *selection sort*.
Per ordinare è possibile usare la funzione **sort**, che ordina in senso ascendente (default) o discendente.

```
>> vec = randi([1, 20],1,7)
vec =
    17  3  9  19  16  20  14
>> sort(vec)
ans =
    3  9  14  16  17  19  20
>> sort(vec, 'descend')
ans =
    20  19  17  16  14  9  3
```

Per una matrice, ogni colonna viene ordinata in modo individuale.
sort(mat,2) ordina sulle righe invece che sulle colonne.

8.8.1 Ordinamento di stringhe

Per ordinare un Cell array di stringhe alfabeticamente si usa **sort**:

```
>> sciences = {'Physics', 'Biology', 'Chemistry'};
>> sort(sciences)
ans =
    'Biology' 'Chemistry' 'Physics'
```

Per una matrice di caratteri, tuttavia, questo non funzionerà perchè sort ordinerà semplicemente ogni colonna. La funzione **sortrows** ordinerà le righe in un vettore di colonne.

8.9 Indicizzazione

Invece che ordinare l'intero vettore ogni volta su un particolare campo, è frequentemente più efficiente creare vettori indice basati sui diversi campi. I vettori indice danno l'ordine in cui il vettore dovrebbe essere attraversato.
Ad esempio vec([3 1 2]) dice "attraversa il vettore in questo ordine: il terzo elemento, poi il primo, poi il secondo".

8.9.1 Indicizzazione in un vettore di strutture

Tabella non ho tempo di farla.
Esempio (iterare attraverso il vettore nell'ordine definito dal vettore codice indice ci):

```
for i = 1:length(parts)
    do something with parts(ci(i))
end
```

Per creare un vettore indice, si usa una funzione sort.

9 Immagini

MATLAB può importare immagini in diversi formati. Le immagini sono rappresentate come matrici mxn in cui ogni elemento corrisponde ad un **pixel**. Tutti gli operatori in MATLAB definiti sulle matrici possono essere usati sulle immagini: +, -, *, /, ^, sqrt, sin, cos, ecc. Ogni elemento che rappresenta un pixel contiene il colore per quel pixel.
I tipi di immagine in MATLAB sono:

- immagini binarie: 0,1
- immagini di intensità: [0,1] o uint8, double, ecc
- immagini RGB: m-by-n-by3
- immagini indicizzate: p-by-3 color map
- immagini multidimensionali: m-by-n-by-p (p è il numero di conche).

9.1 Rappresentazione del colore

Ci sono due vie base per rappresentare il colore in un pixel

- **true color**, o **RGB**, dove i 3 componenti di colore sono salvati (rosso, verde, blu, in questo ordine) → la matrice è mxnx3
- indice in una **colormap**: il valore salvato in ogni elemento della matrice mxn è un intero che si riferisce ad una riga in un'altra matrice chiamata colormap (che ha una grandezza px3 perchè contiene i valori di rosso,verde e blu)

La funzione **image** mostra la matrice dell'immagine usando la corrente colormap:

```
image(mat)
```

La funzione **colormap** può essere usata in due modi: senza argomenti ritorna la colormap corrente, se una matrice px3 viene passata, seleziona quella matrice come colormap corrente.

Esistono diverse colormap incluse, come parula (default), jet, autumn, pink, ecc. Tutte queste hanno 64 colori quindi la dimensione della colormap è 64x3.

9.2 Funzioni utili

La funzione **imread** può leggere un file immagine, per esempio un file JPG. La funzione legge immagini di colore in una matrice 3D:

```
>> myimage1 = imread('xyz.JPG');
```

La funzione image mostra questo:

```
>> image(myimage1)
```

Altre funzioni sono:

- **imshow** mostra un'immagine
- **rgb2gray** converte da RGB in scala di grigio
- **im2double** converte una matrice immagine in **double**

9.2.1 Importazione e esportazione immagini

Leggere e scrivere immagini in MATLAB:

```
>> I=imread('ngc6543a.jpg');
>> imshow(I)
>> size(I)
ans = 479 600 3     (RGB image)
>> Igrey=rgb2gray(I);
>> imshow(Igrey)
>> imwrite(lgrey, 'cell_gray.tif', 'tiff')
```

Alternative a imshow:

```
>>imagesc(I)
>>imtool(I)
>>image(I)
```

9.3 Istogrammi

L'istogramma traccia il numero di pixel nell'immagine (asse verticale) con un particolare valore di luminosità (asse orizzontale).
Algoritmi nell'editor digitale consentono all'utente di regolare visivamente il valore di luminosità di ciascun pixel e di visualizzare dinamicamente i risultati man mano che le regolazioni sono fatte. Miglioramenti della luminosità e del contrasto dell'immagine possono quindi essere ottenute.
Nel campo della visione artificiale, gli istogrammi delle immagini possono essere strumenti utili per la soglia. Poichè le informazioni contenute nel grafico sono una rappresentazione della distribuzione dei pixel in funzione della variazione tonale, gli istogrammi delle immagini possono essere analizzati per i picchi e/ le valli. Questo valore di soglia può quindi essere utilizzato per il rilevamento dei bordi, segmentazione dell'immagine e matrici di co-occorrenza.

- **imhist(I)** calcola l'istogramma per l'intensità di immagine I e mostra il grafico dell'istogramma. Il numero di bins nell'istogramma è determinato dal tipo di immagine.
- **imhist(I,n)** calcola l'istogramma, dove I specifica il numero di bins usato nell'istogramma, n specifica la lunghezza della colorbar mostrata alla base del grafico istogramma.
- **imhist(X,map)** mostra un istogramma per l'immagine indicizzata X; l'istogramma mostra la distribuzione di pixel sopra una colorbar della colormap.

Argomenti di input:

- I: input intensità immagine
- n: numero di
- X: input immagine indicizzata
- map: colormap associata con immagine indicizzata specificata come array p-by-3.

Esempio istogramma immagine:

```
>>I = imread('pout.tif');
>>imhist(I)
>> imhist(I,40)
>>[X,map] = imread('trees.tif');
>>imshow(X,map)
>>Imhist(X,map)
```