Elaborazione delle Immagini

Sara Angeretti
@Sara1798

Fabio Ferrario
@fefabo

2023/2024

Indice

Capitolo 1

Introduzione a MatLab

1.1 Appunti video lezione 2020 da Moodle

1.1.1 Ambienti di lavoro

Command Window e Workspace

- Command Window: finestra di comando, dove andiamo effettivamente a scrivere i comandi.
- Workspace: variabili in memoria, vediamo le variabili che abbiamo istanziato, mi dà un feedback immediato di quello che ho fatto.

Variabili

```
>> pippo = 10
pippo =
    10
>>
```

Questo vuol dire che istanziare una variabile assegnandole anche un valore numerico, dopo che ho premuto invio mi viene creata in memoria la variabile che ho creato (lo vedo nel workspace che l'ho creata). Però scrivendo così mi viene anche stampato a schermo il comando che ha eseguito.
Nel workspace vedo che ho una variabile chiamata pippo che ha valore 10, size 1x1 e class double.
Questo vuol dire che ha dimensione 1x1, cioè che è una singola cella (MatLab lavora con dati che sono matrici). Inoltre, double è il valore base, di default, quasi tutto viene fatto in virgola mobile (ci possono essere forzature se necessario).

Ci sono operazioni più complesse però di semplici assegnamenti, per cui potrebbe tornare utile evitare di stampare a schermo, perché non è assolutamente necessario vedere live l'output del comando che viene eseguito. Perciò:

```
>> pluto = 10;
>>
```

1.1.2 Lista di comandi utili

¿¿ clear: pulisce il workspace, cancella tutte le variabili che ho istanziato.

¿¿ clc: pulisce il command window, cancella tutti i commenti che ho scritto.

freccia su tastiera: va a riprendere l'*history* dei miei comandi nella command window, che apre in una finestrella pop-up.

1.1.3 Operazioni matematiche

Addizione

```
>> a = 2;
>> b = 1;
>> c = a + b;
```

Sottrazione

```
>> a = 2;
>> b = 1;
>> c = a - b;

ans =

    1
```

Prodotto

```
>> a = 2;
>> b = 1;
>> c = a * b;

ans =

    2
```

Divisione

```
>> a = 2;
>> b = 1;
>> c = a / b;

ans =

    2
```

Elevamento a potenza

```
>> a = 2;
>> a^3;

ans =

    8
```

MatLab richiede che il valore di un'operazione venga associato ad una variabile. Se non la creo io, lo fa MatLab per me (qua la variabile "`ans`"). Visibile nel *Workspace*.
Questa variabile può essere riutilizzata.

```
>> a + ans;

ans =

    10
```

Però ocio che ad ogni operazione verrà sovrascritta.

Altre operazioni

Si può operare in tanti modi con gli scalari su MatLab; per vedere le operazioni possibili, si deve consultare la documentazione.

1.1.4 La documentazione

1. Tasto "*Help*" in "Resources"
2. In Command Window possiamo chiederlo direttamente a MatLab:

- ▷ comando "`help`", mi dà dei suggerimenti sull'ultima operazione eseguita.
- ▷ comando "`help max`", (dove "`max`" è un esempio di operazione che dà in output il massimo numero in un array di numeri dati in input) mi dice quale è la sintassi con tutte le sue alternative dell'operazione richiesta.
- ▷ comando "`lookfor max`", (dove "`max`" è un esempio di parte del nome di un'operazione che non ci ricordiamo nella sua interezza) mi dà una lista di possibili operazioni che potrebbero essere quella che stiamo cercando e che contengono il pezzo di nome che gli ho assegnato.

1.1.5 Gli array

Sintassi e operazioni sui vettori

Vettori riga: i vettori sono rappresentati come concatenazione di valori, secondo la sintassi:

```
>> v = [1,2,3,4,5,6];
>> v

v =

    1   2     3     4     5     6

>>
```

Nel workspace vedremo:

Name	Value	Size	Class
`v`	`[1,2,3,4,5,6]`	`1x6`	`double`

In questo caso è un **vettore riga**.
Le *parentesi quadre* si usano per **concatenare** i valori, le *virgole* per separare i valori in un *vettore riga*, i *punti e virgola* per separare i valori in un *vettore colonna*.

Vettori colonna: se invece volessimo un **vettore colonna**:

```
>> w = [10;20;30;40];
>> w

w =

    10
    20
    30
    40

>>
```

Nel workspace vedremo

Name	Value	Size	Class
`w`	`[10;20;30;40]`	`4x1`	`double`

N.B.: sono ***indicizzabili***. Per indicizzare si usano le *parentesi tonde* e, cosa importantissima, *in MatLab* ***non esiste l'indice 0****, perciò si parte da 1 e non da 0.* Se si scrive lo 0 come indice, dà errore.
Se volessimo ad esempio il terzo elemento del vettore `v`:

```
>> v(3)

ans =

    3

>>
```

Assegnare un valore: questa indicizzazione la possiamo usare anche in assegnazione:

```
>> v(3) = 100;
>> v

v =

    1   2     100     4     5     6
```

```
>>
```

Posso lavorare anche su *serie di cellette* invece di una celletta singola:

```
>> v = (1:3);

ans =

1   2     100

>> v = (1:3);

ans =

    4     5     6

>>
```

Se volessi, senza sapere quanto è lungo un vettore (c'è un modo per saperlo, ma non lo vediamo ora), prendere tutti gli elementi da un certo indice fino alla fine, posso fare:

```
>> v(4:end)

ans =

    4     5     6

>>
```

Scoprire la lunghezza di un vettore: con il comando:

```
>> size(v)

ans =

    1   6
```

```
>>
```

Ovvero una riga e 6 colonne. La convenzione (è sempre così) è prima righe e poi colonne, perciò se volessi conoscere solo le righe:

```
>> size(v,1)

ans =

    1

>>
```

se volessi conoscere solo le colonne:

```
>> size(v,2)

ans =

    6

>>
```

Somma: come sappiamo da GAL, se sommo due vettori devono avere la stessa dimensione, altrimenti non posso sommarli.

Prodotto matriciale: come sappiamo da GAL, se moltiplico due vettori devono avere gli stessi indici interni, altrimenti non posso moltiplicarli. Es.: noi abbiamo `v 1x6` e `w 4x1`, non posso moltiplicarli perché $6 \neq 4$. Una cosa che potrei fare per ovviare il problema sarebbe selezionare un sotto vettore da `v` che abbia la stessa dimensione di `w`:

```
>> v(1:4)*w

ans =

    3210
```

```
>>
```

Per chi non ha ancora visto GAL, questo è un prodotto scalare, ma perché `ans` vale `3210`? Praticamente ho due matrici $m \times p$ e $p \times n$, la dimensione della matrice prodotto sarà $m \times n$. Quindi qua ho un vettore riga 1×4 e un vettore colonna 4×1, il prodotto sarà un vettore riga 1×1, che è quello che vediamo in `ans`. Inoltre il prodotto fra:

una matrice			e una matrice	
a_1	a_2	a_3	b_1	b_2
a_4	a_5	a_6	b_3	b_4
			b_5	b_6

mi dà una matrice 2×2 con i valori:

$a_1 * b_1 + a_2 * b_3 + a_3 * b_5$	$a_1 * b_2 + a_2 * b_4 + a_3 * b_6$
$a_4 * b_1 + a_5 * b_3 + a_6 * b_5$	$a_4 * b_2 + a_5 * b_4 + a_6 * b_6$

È quello che succede se:

```
>> z = v(1:4)

z =

    1   2     100     4

>> w*z

ans =

    10   20     1000     40
    20   40     2000     80
    30   60     3000     120
    40   80     4000     160

>>
```

Prodotto puntuale : .* (punto per punto), ovvero moltiplica elemento per elemento, pixel per pixel, lasciandoli al loro posto.

Valore massimo: `max(v)` mi ritorna il valore massimo fra i numeri del vettore. Per visualizzare le varianti, basta scrivere `max` e aprire la parentesi tonda sinistra e in sovrimpressione uscirà la lista delle varianti. In questo caso:

```
>> max(v)

ans =

    100

>>
```

Somma dei valori: `sum(v)` mi ritorna il valore totale della somma dei numeri del vettore. Questo, come il comando precedente, ha un comportamento diverso se si tratta di vettori o di array. In questo caso:

```
>> sum(v)

ans =

    118

>>
```

Creare matrici di numeri random: `randi(r, l, c)` mi ritorna un array di `l` righe e `c` colonne di numeri scelti randomicamente in un range ampio `r`.

```
>> v = randi(5,1,10)

v =

    1   5   1   4   5   5   1   2   2   5

>>
```

Trovare un valore uguale ad un numero dato in un array: mi ritorna un vettore booleano che MatLab chiama *vettori logici* che mi dice dove la condizione che ho espresso è vera, ovvero il valore del vettore è uguale al numero dato quando c'è 1.

```
>> v == 5

ans =

    1x10 logical array

    0   1   0   0   1   1   0   0   0   1

>>
```

Non è male perché è una semplice riga di istruzione, mi evita cicli espliciti e altra roba.

Trovare i valori maggiori di un numero dato: mi ritorna un *vettore logico* che mi dice dove la condizione che ho espresso è vera.

```
>> v>3

ans =

    1x10 logical array

    0   1   0   1   1   1   0   0   0   1

>>
```

Sono operazioni molto utili. E combinabili!

```
>> sum(v>3)

ans =

    5

>>
```

Mi conta quanti sono gli 1 del vettore logico.
Posso usare variabili di appoggio:

```
>> x = v>3

x =

    0   1   0   1   1   1   0   0   0   1

>> v

v =

    1   5   1   4   5   5   1   2   2   5

>>
```

Selezionare solo alcuni valori: `>> v(x)`

```
ans =

    5   4   5   5   5

>>
```

Praticamente mi seleziona solo i valori di `v` che hanno 1 nel *vettore logico* `x`.
Molto utile per creare le maschere! Posso indicizzare il mio vettore anche usando un altro vettore di numeri:

```
>> v([1,3,4,6])

ans =

    1   1   4   5

>>
```

Questo invece mi ritorna i valori di `v` che mi interessano, che ho chiesto esplicitamente.

Posso selezionare solo i valori che hanno 1 nel vettore logico anche in un altro modo:

```
>> find(x==1)

ans =

    2   4   5   6   10

>>
```

Questo mi ritorna gli *indici* delle cellette di `x` che mi soddisfano questa condizione.
Posso ovviamente poi usarlo per ottenere i corrispondenti valori di `v`.

```
>> v(find(x==1))

ans =

    5   4   5   5   5

>>
```

Notare che mi produce lo stesso risultato della sua variante precedente, `v(x)`. Queste due istruzioni sono equivalenti.

Sintassi e operazioni sulle matrici

Possiamo istanziare una matrice andando a memorizzare tutti i valori che vanno salvati nelle righe e nelle colonne. Le operazioni sono pressoché le stesse che abbiamo visto per i vettori, solo che ora abbiamo due dimensioni e quindi due indici, uno per le righe e uno per le colonne.

Riempire una `3x2` a mano:

```
>> a=[1,2;3,4;5,6]

a =

    1   2
    3   4
    5   6
```

```
>>
```

Trovare il valore di una specifica celletta: ad esempio terza riga prima colonna,

```
>> a(3,1)

ans =

    5

>>
```

Visualizzare un'intera riga: ad esempio la seconda

```
>> a(2,1:end)

ans =

    3   4

>>
```

In alternativa, ":" indica "tutti gli elementi", quindi:

```
>> a(2,:)

ans =

    3   4

>>
```

Visualizzare un'intera colonna: funziona come con le righe, ad esempio la prima

```
>> a(:,1)

ans =

    1
    3
    5

>>
```

Matrice trasposta: `>> a'`

```
ans =

    1   3   5
    2   4   6

>>
```

Creare matrici di comodo: utili per creare matrici su cui computare operazioni:

```
>> zeros(3,3)

ans =

    0   0   0
    0   0   0
    0   0   0

>> ones(5,5)

ans =

    1   1   1   1   1
    1   1   1   1   1
    1   1   1   1   1
    1   1   1   1   1
    1   1   1   1   1
```

```
>>
```

Creare matrici randomiche: abbiamo visto prima il comando rand, che mi crea numeri randomici double. Se voglio creare una matrice intera, posso usare il comando randi, che mi crea una matrice intera randomica.

```
>> m=rand(5,5)

m =

    0.4314  0.1361  0.8530  0.0760  0.4173
    0.9106  0.8693  0.6221  0.2399  0.0497
    0.1818  0.5797  0.3510  0.1233  0.9027
    0.2638  0.5499  0.5132  0.1839  0.9448
    0.1455  0.1450  0.4018  0.2400  0.4909

>>
```

Creare matrici inverse: `>> inv(m)`

```
ans =

   -9.6261   -0.7953   -70.5587    84.5354   -24.6859
    5.3676    1.3707    43.2380   -50.8728    13.6995
    7.3422    0.6401    42.8965   -51.9585    14.8125
   -1.0754    0.7834     6.1327   - 9.5106     7.8621
   -4.2154   -1.0759   -29.9599    37.1396   -10.6575

>>
```

Determinante: restituisce il suo valore

```
>> det(m)

ans =

    -0.0014
```

```
>>
```

Diagonale: riporta i valori posizionati sulla diagonale principale della matrice

```
>> diag(m)

ans =

    0.4314
    0.8693
    0.3510
    0.1839
    0.4909

>>
```

Sottomatrici: riporta solo i valori della sottomatrice scelta, ad esempio quella fra seconda e terza riga, terza e quarta colonna

```
>> m(2:3,3:4)

ans =
0.6221  0.2399  0.0497
0.1818  0.5797  0.3510  0.1233

>>
```

1.2 Appunti Lab1 13 Ottobre 2023

1.2.1 Il negativo di un'immagine

Serve per rimappare un'immagine. Lo faccio sottraendo la prima immagine a 255 e asegnando il tutto ad una seconda immagine.

```
>> im1 = imread('path\img.format');
>> im2 = 255 - im1;
>> imshow(img2);
```

1.2.2 Mostrare a schermo due immagini

```
>> figure(i);
>> subplot(1,2,1);
>> imshow(im1);
>> subplot(1,2,2);
>> imshow(im2);
```

figure; : crea una finestra, quando ne serve più di una si aggiunge `(i)` con un indice a scelta.
Se non passiamo niente in input, MatLab indicizza da solo, altrimenti glielo diciamo noi l'indice.
Rendo attiva la finestra richiamandola con l'indice scelto.

subplot(r, c, q); : `subplot(n° righe, n° colonne, n° quadrante)`, rende attiva la `figure` e divide la finestra in riquadri (puoi fare più righe ($r > 1$) e più colonne ($c > 1$)) e `q` dice in quale quadrante (della riga scelta con `r`) mettere l'immagine.

Poi abbiamo creato `im3` come somma di `im1` e `im2`, cosa esce? Un'immagine tutta bianca, perché si sommano tutti i valori matematici e sarà tutto 255. Poi, per mostrarla a schermo, se faccio solo `imshow(im3)` andrà a sovrascrivere quella negativa, perché non ho specificato che serve una nuova figure e quindi usa l'ultima attiva. Serve `figure;` per averne una nuova.

1.2.3 Pulizia dello schermo

Ogni volta che eseguo lo script mi si aprono finestre di figure.

`close all;` : chiude in automatico tutte le finestre.

`clear all;` : cancella in automatico tutte le variabili.

1.2.4 Modificare size immagini

```
imresize(clouds, size(moon))
```

1.2.5 Le maschere binarie

Mi individuano una zona di interesse (in questo caso "dove vivono le nuvole") da togliere poi all'immagine originale.
Come? `treeshold` mi ritorna un vettore logico, che posso usare come maschera binaria. Per ora la soglia la troviamo a mano.

Abbiamo provato T = 0.5, ma non bastava. Abbiamo abbassato a 0.25, che poteva andare per quanto riguarda la zona di interesse, ma abbiamo abbassato ancora a 0.125 che mi ha preso ancora un po' di pixel.
Spoiler alert: alla fine abbiamo rimesso 0.25 perché la maschera aveva preso troppi pixel scuri e quando si zoommava sulla foto finale era tutta rovinata con bordi scuri etc.
Abbiamo provato 0 ed è diventato troppo, ha mostrato un effetto quadrettato che mi mostra artefatti/rumore che è tipico della compressione JPEG.
0.125 va più che bene, poi la usiamo per togliere un pezzetto di immagine da quella della Luna.

$1 - mask_clouds$ mi ritorna la maschera binaria invertita, ovvero dove c'è 1 mette 0 e viceversa.

$.*$ moltiplicazione punto punto, dove nella maschera invertita ho 0, si azzera il pixel corrispondente nell'immagine.

Se volessi una maschera più smooth? Così che si nasconda il passaggio tra nuvola e sfondo della nuvola? Applico un filtro di smooth sull'immagine delle nuvole. Non tolgo totalmente il rumore, ma lo riduco localmente.

Immagini double

Prima mi sono dimenticata di segnare che spesso serve avere le immagini in forma double, si fa tipo

```
moon = im2double(imread("Lab1\moon.jpg"));
```

usando quindi il comando

```
im2double();
```

che converte l'immagine da *uint8* in *double*.

1.2.6 Gli istogrammi

`imhist();` mi ritorna l'istogramma dell'immagine.
Dall'istogramma dell'immagine del cavallo vedo che è **ben contrastata**, ovvero uso tutti i valori del range di valori dispobile.
`histeq();` mi ritorna l'immagine equalizzata.

1.2.7 Gamma correction

`imadjust();` mi ritorna l'immagine con la gamma corretta.
È un mapping non lineare, che mi permette di rimappare i valori di un'immagine.

1.2.8 Compiti a casa - 1

Sort of assignment, sono 4 e danno fino a 2 punti in più all'esame (quindi se ne consegno 2 è solo 1 punto in più). Per dire, il primo chiede di scrivere una funzione `myhistogram` che calcola l'istogramma di una immagine a livelli di grigio. Noi sappiamo che un istogramma cumulativo (uint8 tra 0 e 255 monocanale) è un vettore che va da 0 a 255 e mi conta le occorrenze di ogni valore.