MatLab:

MatLab può immagazzinare sia variabili singole (es. 9, a, L, pippo, valori booleani cioè veri/falsi) che vettori o matrici; questa capacità è nota come "casting dinamico delle variabili".

Gestione dei valori logici booleani su MatLab:

Una variabile logica può immagazzinare due valori:

-Vero → tramite il simbolo 1 (ma in realtà da qualsiasi valore numerico diverso da 0);

-Falso → tramite il simbolo 0.

Gli operatori relazionali in MatLab sono rappresentati da:

-Minore < → <

-Maggiore > → >

-Maggiore uguale ≥ → >-

-Minore uguale ≤ → <-

-Uguale = → ==

-Diverso ≠ → ~ -

-Somma tra vettori e matrici → +

-Moltiplicazione tra vettori (elemento per elemento) → *

-Moltiplicazione tra matrici (elemento per elemento) → .* per poterlo usare le dimensioni delle matrici devono concordare.

Gli operatori logici in MatLab sono rappresentati da:

-And → **&**

-Or → |

-Not → ~ (tilde)

Se voglio elementi che sono compresi tra due numeri, non scrivo per es. 2 < a < 4 , ma devo usare gli operatori logici per ottenere un predicato composto, del tipo (a > 2) & (a < 6). Se ho **un intervallo (valori compresi) uso &**; se ho **due intervalli disgiunti (minore di/maggiore di) uso Or**.

Funzioni in MatLab:

-**logical(x)** → converte gli elementi del vettore x in valori logici, dando ad ogni 0 il corrispettivo valore che significa falso, e dando 1 a qualsiasi altro valore diverso da 0, che significa vero;

-**size(x)** → mi dà le dimensioni di una matrice (= n° di righe e colonne), ma posso usarlo anche con i vettori;

-**length(x)** → mi dà la lunghezza di un vettore (= n° di elementi del vettore), questa funzione non si usa con le matrici;

-**trasposta** (cioè l'apice, x') → mi fa fare l'operazione inversa (es. posso scambiare le righe con le colonne e viceversa);

-**exist(x)** → mi dice se la variabile esiste, cioè se è stata instanziata. Affinché funzioni occorre prima che la variabile sia stata instanziata tramite il comando **input**, il cui argomento deve essere una stringa/carattere di vettori;

-**isempty(x)** → indica se una variabile è vuota, cioè con zero valori;

-**all(x)** → prende un vettore e determina se tutti i suoi elementi sono veri (ovvero se sono diversi da zero);

-**sum(x)** → mi permette di **contare gli elementi** di un vettore/matrice (es. quanti numeri 4 ho in quel vettore/matrice);

-**char(x)** → mi permette di riconvertire in vettori di caratteri la stringa che avevo instanziato in x;

-**char(a, b)** → mi fa mettere i vettori a e b in colonna;

-**string(b (2:4))** → data una sequenza di caratteri mi crea la stringa corrispondente;

-**lower(x)** → mi rende tutta la stringa x in caratteri minuscoli;

-**upper(x)** → mi rende tutta la stringa x in caratteri maiuscoli;

-**strcmp(a, b)** → mi permette di comparare due stringhe: se sono uguali avrò come risultato 1 se non lo sono avrò come risultato 0;
-**strrep(a, 'a', 'b')** → è la funzione "string replace" e prende 3 elementi: la stringa a, l'elemento da sostituire ('a') e l'elemento con cui deve essere sostituito ('b');
-**findstr(b, 'i')** → mi permette di cercare una sottostringa ('i') all'interno della stringa (b);
-**sprintf** → mi permette di inserire stringhe di caratteri e di combinarle tra loro per formare delle frasi, in quanto ammette dei segnaposto (%d, %s etc.). È un comando formattato e spesso si usa nei cicli, quando dobbiamo stampare una sequenza di nomi. Può essere usato anche per creare una variabile stringa;
-**disp(x)** → fa la stessa cosa di sprintf ma non ammette i segnaposto (%s, %d etc.) e mi permette di **stampare**/mostrare o una singola stringa senza variabile all'interno, o una singola variabile. Vuole solo un singolo argomento;
-**input('…')** → mi permette di inserire un ingresso. Tra apici va inserita la domanda, che verrà riportata sotto al comando; qui poi possiamo aggiungere un ingresso. MatLab in automatico prenderebbe i numeri, per cui devo specificare come secondo argomento se voglio una stringa ('s') o un singolo carattere ('c');
-**isnan** → mi permette di individuare gli elementi NaN in una matrice/vettore tramite un vettore di booleani (non ho un certo dato laddove avrò 1, che indica presenza di NaN);
-**mean** → mi fa fare la media degli elementi di un vettore. Ricordarsi che **la media prende le somme parziali degli elementi di un vettore** nella scrittura dello script;
-**rmfield** → mi consente di rimuovere i campi in un dato strutturato;
-**uimport** → mi permette di importare dati dall'esterno;
-**plot([x,y])**→ realizza il disegno; ha come input due vettori numerici che rappresentano i valori sulle x e i corrispettivi valori sulle y, prima instanzio le variabili x e y. Se voglio rappresentare insieme due funzioni, aggiungo una coppia di vettori x,y oppure uso il comando **hold on** affinché venga aggiunto a quanto già rappresentato, se poi voglio toglierlo uso **hold off**. I comandi hold on/hold off vanno aggiunti alla fine, dopo le coppie di valori;
-**subplot(Nrows, Ncolumns, position)** → mi permette di visualizzare più grafici nello stesso momento. La posizione è quella della slot, ovvero il quadrato dove andrà il plot;
-**bar(rand(x,y), 'stacked')** → mi fa disegnare grafici a barre cumulativi. Il comando rand mi permette di generare numeri random, x e y mi danno le dimensioni della matrice (ma al posto di rand posso inserire direttamente due valori), 'stacked' indica che i valori sono messi uno sopra l'altro. Es. rand(1,5) ho una matrice 1x5;
-**rand** → mi consente di generare numeri random da 0 a 1;
-**randi(n, i, j)** → per generare casualmente numeri interi: **n** mi dice **fino a quale numero vogliamo generare degli interi**, **i e j sono il numero di righe e colonne di una matrice** che viene generata e riempita con questi interi. Se ho una stringa di testo es. a = 'Mario' , 'Luigi', 'Davide' per sceglierne uno casualmente avrò >> a(randi(3)) ;
-**dev** → mi permette di stabilire la deviazione per ciascuna barra nei grafici a barre non cumulativi;
-**randi([n, m], i, j)** → mi fa generare numeri interi in un dato intervallo: n e m costituiscono l'intervallo di numeri entro cui vogliamo generare gli interi, i e j mi danno il numero di righe e di colonne di una matrice che viene riempita con gli interi generati. Es. randi([-2,3], 4,8) → matrice 4x8 di numeri $-2 < x < 3$;
-**str2num(x)** → "string to number", è usato per passare da stringhe a numeri. Se scrivo str2num('ciao') la risposta sarà [ ] ovvero una risposta vuota, inesistente perché non è un numero;
-**for** → **sintassi per ottenere un ciclo sia di elementi che appartengono a un insieme che di elementi in un intervallo**;
- **%** → per commentare una parte del codice;

-**while** → **sintassi per ottenere un ciclo nel caso della media di un vettore o della creazione di un menù**. while necessita dell'incremento della variabile di ciclo che ho instanziato alla fine del ciclo;
-**ismember** → prende due elementi e guarda quante volte il primo è contenuto nel secondo;
-**break** → forza l'uscita da un ciclo, blocca quello più interno (es. se ne ho 3, termina l'ultimo);
-**function[output] = nomefunzione(input)** → es. function[max, min] = prima_funzione(x) → prima_funzione prende un vettore (x) e restituisce il max e il min valore presente. Devo sempre instanziare gli output seguiti dal punto e virgola affinché MatLab non li stampi;
-**help** → mi dà il commento che ho scritto sotto la funzione che vado a richiamare. Es. se richiamo Luigi, mi dà il commento scritto sotto alla funzione Luigi;
-**type(nome file)** → mi fa vedere il contenuto di un file;
-**readfile(inputArg1)** → mi permette di leggere linea per linea di un file, e inputArg1 è il file che deve essere aperto;
-**fopen(inputArg1)** → comando che mi consente di aprire un file. Include opzioni aggiuntive come: **r** indica che apro il file per sola lettura, **r+** indica che apro il file sia in lettura che in scrittura e senza che si cancelli il suo contenuto, **w** indica che apro un file per scrittura e se non c'è me ne crea uno nuovo, mentre se c'era qualcosa già scritto sopra me lo cancella, **w+** indica che apro/creo un file per lettura e scrittura, **a** indica che apro/creo il file in modalità append, cioè posso scrivere sul file alla fine, e **a+** indica che apro/creo un file per append e lettura. Quando apro un file l'handler è al 1° elemento (inputArg1), se l'apertura fallisce ottengo un intero (-1) che non è l'inizio del file. Inoltre, quando apro un file è necessario inserire nello script il controllo della sua esistenza (es. if x == **-1** sprintf(' %s non esiste', inputArg1);
-**feof** → comando che mi dice se ho raggiunto l'eof (end of line) di cui x è puntatore (= il mio file);
-**fgetl** → prende la linea dal punto in cui si trova il puntatore fino alla sua fine, portando il puntatore all'inizio della linea successiva ammesso che non sia l'ultima del file;
-**function writefile(inputArg1, inputArg2)** → per scrivere in un file: può avere vari argomenti di cui il primo è il nome del file. Per indicare un matrice, che potrebbe essere inserita come inputArg2, devo sapere le sue dimensioni, pertanto avrò: **[nrows, ncols] = size(inputArg2)**;
-**fprintf(handler, format, text)** → mi permette di scrivere in un file a partire dall'handler del file che voglio scrivere (x). Il secondo ingresso è dato dal formato con cui si vuole scrivere nel file (es. una stringa di testo, uno o più numeri) e pertanto sarà dato da %s o %d (se sono più di uno vuol dire che ho più formati, oppure più argomenti con lo stesso formato: se ho %d %d %d vuol dire che ho una matrice con 3 colonne), mentre il terzo ingresso è dato dal text, ovvero da cosa voglio scrivere, indicato con una variabile ad esempio y, instanziata prima (es. y = input('inserire il testo', 's');
-**sound(tone_values,frequence)** → mi permette di generare suoni; prende di default due argomenti: un vettore di valori (tone_values) compreso tra -1 e 1, e il campionamento (frequency), cioè con che frequenza (**sr = simple ratio**) devono essere suonati i toni che abbiamo passato. La sr è il campionamento dell'onda: indica pertanto quante volte al secondo vado a leggere il valore dell'onda, ovvero quanti elementi del vettore dò in output ogni secondo, e il suo valore base è **44100**. Tale valore può cambiare e teoricamente potrebbe essere qualsiasi, tuttavia sotto i 3000/4000 il suono degenera. La frequenza può essere data anche da **sr*d**, ovvero da sr moltiplicato per la durata del suono, oppure da un intervallo di valori (es. 0,2*pi*f*d, sr*d) dove sr*d indica l'equispaziatura dei valori. Se tone_values = 20 e sr = 10, significa che ogni secondo MatLab produce 10 elementi del vettore, quindi il suono durerà 2 secondi;
-**linspace(0,d, sr*d)** → mi permette di avere una sequenza equispaziata di un determinato intervallo (es. t = linespace(0,d, sr*d) se d = 1 significa che ottengo un'equispaziatura di 44100 valori fra 0 e 1). Rappresenta quindi il vettore di elementi della mia onda e devo instanziarlo se voglio che suoni qualcosa;

-**uint8** → indica 8 bits di numeri che vanno da 0 (nero) a 255 (bianco) per i pixel che rappresentano diverse intensità di grigio;
-**mycolor** → comando che mi permette di creare la mia palette. Per es. avrò mycolor = [0 0 0; 0.8 0.2 0.1; 1 1 0.8], i valori devono essere compresi tra 0 e 1 e poi colormap(mycolor);
-**colormap** → mi mostra la matrice della palette table, i cui **valori sono gli RGB che vanno da 0 a 1** (non da 0 a 255) e mi determinano l'intensità dei colori rosso, verde e blu: es. per mostrare la mia palette "my color" dovrò scrivere colormap(mycolor);
-**image(a)** → mi fa visualizzare l'immagine a che ho importato con imread, cioè la matrice;
-**colormapeditor** → mi permette di accedere alla mappa dei colori per cambiare i 64 colori della palette affinché possa personalizzarli;
-**variable = imread('filename', 'file type')** → mi permette di leggere un'immagine dall'esterno. Vengono definite matrici diverse a seconda del tipo di immagine, cioè della loro estensione. variable corrisponde a [A, B] dove in A va la matrice che contiene l'immagine, e in B la palette di colori se presente. Es. A = imread('carlino.jpg', 'jpg') → ottengo una matrice numerica che rappresenta l'immagine, poi visualizzo l'immagine con image(A);
-**imwrite(variable; 'file name', 'file type')** → per salvare la mia immagine. "variable" è la variabile che contiene la mia immagine;
-**rgb2gray()** → mi permette cambiare la mia immagine da rgb a toni di grigio, e avrà per ogni pixel valori compresi tra 1 e 256;
-**imshow(A, colormapofA)** → mi fa vedere la colormap di A che ho ottenuto tramite imread, ovvero prendendo tutti i colori presenti nell'immagine;
-**colormapgray(numero)** → per ottenere tonalità di grigio in un determinato range e quindi ampliare l'attuale palette;
-**cast(ciò che deve essere trasformato, tipo in cui deve essere trasformato)** → mi fa trasformare la tipologia degli elementi dell'immagine, es. per avere wind in uint8 anziché in double;
-**line([vettore 1],[vettore 2])** → mi crea una linea spezzata in base ai punti dei vettori: il primo è inerente alle x dei punti, il secondo alle y;
-**fill([vettore 1],[vettore 2], 'r')** → per creare un rettangolo colorato di rosso con quei punti: r sta per 'red'.

Psychtoolbox, comandi:
-**try/catch/end** → è un costrutto che può essere inserito in una procedura e permette a MatLab di eseguire tutti i comandi compresi tra try e catch. È molto utile poiché mi permette di vedere se ci sono errori, e qualora ci fossero la procedura viene interrotta;
-**Screen('SubFunctionName',parameter1, parameter2…)** → è un operatore che "crea" e permette di visualizzare quanto creato. Come primo argomento ha una sotto-funzione, cioè quello che vogliamo far fare alla funzione Screen, e poi i parametri associati alla funzione;
-**Screen('SubFunctionName?')** → per vedere l'help su ciascuna funzione;
-**OpenWindow** → per aprire una finestra, ovvero una figura (utile per usare la funzione Screen per disegnare figure). Per farlo sono necessari **tre step**: apro una finestra (figura), disegno/modifico qualcosa lì, chiudo la finestra. OpenWindow sarà il primo argomento di Screen; il primo parametro poi sarà un numero (di default è 0, e rappresenta lo schermo principale) che indica su quale dispositivo di output voglio che la finestra venga aperta (= su quale schermo se ne ho più di uno). Dopo lo zero avrò una tripletta rgb che mi dà il colore di sfondo della mia finestra (che è un rettangolo). Infine ho l'area in cui voglio settare la finestra; se non la specifico viene considerata finestra l'intero schermo. La funzione OpenWindow poi ritorna un puntatore alla finestra dello schermo che ho indicato e mostra le coordinate in pixel della finestra che ho aperto [0,0 ; x,y], ovvero quelle dell'angolo in alto a sx (0,0) e di quello in basso a destra (x,y). Come ultima cosa

devo chiudere la finestra con la funzione **Screen('CloseAll')** per chiudere tutte le finestre oppure **Screen('Close',nomefinestra)** per chiuderne una specifica.
Es. **[myscreen, rect] = Screen('OpenWindow', 0,[0,255,0])** → ho creato una finestra verde (rosso e blu = 0) della stessa dimensione dello schermo (dopo la tripletta di colori non ho indicato alcuna area, cioè *rect*). Il nome della mia finestra è myscreen; rect sono i quattro numeri 0,0 e x,y : i due angoli del mio rettangolo (finestra), quindi mi dà le dimensioni del mio rettangolo (o del mio schermo se non stabilisco un'area (un rect)).
Myrect = [10, 20, 150, 250] → indica dove vado a creare la finestra.
Screen('OpenWindow',0, [], Myrect) → mi viene aperta una finestra rettangolare bianca in cui l'angolo in alto a sx è nel punto 10,20 del mio schermo e quello in basso a dx nel punto 150,250. rect della funzione precedente che indica le dimensioni della finestra che apro, allora sarà per le x 140 (150 – 10) e per le y 230 (250 – 20).
La funzione **Screen('Flip',windowPtr)** mi **mostra** ciò che c'è nel backbuffer, ovvero quello che ho creato all'interno della mia finestra cancellando quello che c'era precedentemente, anche se esiste un comando per congelare ciò che era già esistente e per sovrapporvi le cose create dopo.
N.B: la funzione **Screen('FillRect',pippo,[255,0,0], minnie)** → è importante perché **mi dà il background**: mi fa mettere minnie dentro pippo;
-**KbWait** → mi permette di fermare l'esecuzione di uno script finché non viene pigiato un tasto qualsiasi della tastiera;
-**Pause( )** → mi permette di congelare l'esecuzione dello script per un determinato tempo dato in secondi;
-**Scren('DrawText', windowPtr, text [, x] [, y], [, color] [, …])** → per scrivere il testo nella finestra aperta con Screen: lo inserisco nella variabile text, x e y sono le coordinate dell'angolo in alto a sx del rettangolo immaginario contenente il testo, poi avrò anche le coordinate finali del testo per vedere dove finisce. DrawText mi prende stringe di caratteri (scritti con un solo apice, se ho vettori di stringhe devo convertirli con la funzione char); windowPtr è la finestra in cui voglio scrivere. Infine color indica il colore del mio testo. N.B: la sotto-funzione DrawText richiede un vettore di caratteri, per cui se ho delle stringhe devo usare char per convertirle;
-**Screen('TextStyle', pippo, n)** → indica che lo stile di testo nella finestra pippo è n, che è un n° composto tra 0 e 7 e mi indica varie tipologie di testo (es. normale, sottolineato, barrato etc.);
-**Screen('TextFont', pippo, Times New Roman)** → indica la tipologia di carattere che voglio inserire nella finestra;
-**Screen('TextSize', pippo, 36)** → per settare le dimensioni del testo nella finestra pippo;
-**DrawTexture** → mi permette di creare una finestra in cui posizionare l'immagine che poi visualizzerò; per usarla sono necessari tre passaggi: 1) carico l'immagine con imread; 2) creare una **texture** (**pic**) dell'immagine; 3) mostrare l'immagine. **N.B**: Devo anche creare il rettangolo che conterrà la mia immagine, e deve avere le stesse dimensioni di essa (es. r = [0, 0, size(A1), size(A2)]), apro la finestra con OpenWindow e vi centro il r;
-**MakeTexture** → mi permette di convertire un'immagine in texture indicando dove questa texture va fatta e quale immagine la compone: **pic = Screen('MakeTexture',pippo,A)** indica che voglio la pic (texture) nella finestra pippo e dell'immagine A;
-**AlignRect** → mi dice dove devo allineare il mio rettangolo rispetto ad un secondo rettangolo, basandosi sulle x (righe) e y (colonne);
-**WaitSecs (n)** → mi permette di gestire i tempi di esecuzione di un programma/persona. E' simile al comando Pause: mi consente di stoppare il programma per n secondi;
-**GetSecs gets** → mi permette di gestire i tempi di esecuzione di un programma/persona e mi consente di prendere il tempo che è intercorso fra l'esecuzione del comando e l'avvio del computer;
-**PsychPortAudio()** → è un comando che serve per generare suoni e riesce bene a sincronizzarli, è

più versatile di Sound. Per usarlo devo aprire un canale con la funzione **'Open'**, poi avviarlo e infine chiuderlo. La sua sintassi può comporsi di molti elementi:
mychannel = PsychPortAudio('Open' [, deviceid] [, freq] [, channels]);
Indicano rispettivamente in quale canale il suono deve essere prodotto (se non viene specificato, il device usato è quello di default, solitamente lo speaker o le casse), qual è la sua sample ratio e qual è il n° dei canali in cui vogliamo suonare (devo specificare se mono (1) o stereo (2)). Si possono anche aggiungere altre caratteristiche (vedi p. 53). Dopo creo il mio vettore di interi (beep) attraverso **'MakeBeep'**, che genera dei valori che mimano l'onda del mio suono.
mybeep = MakeBeep(frequency, duration_in_secs, sample ratio) → **MakeBeep** crea in automatico un vettore di interi (double).
Per poter suonare la mia onda (il mio beep) devo inserirla nel canale con **'FillBuffer'** :
PsychPortAudio('FillBuffer', mychannel, [myBeep; myBeep]);
Dove [myBeep; myBeep] è una matrice 2 x myBeep, ovvero sample ratio. Per suonarla uso **'Start'** :
PsychPortAudio('Start', mychannel, repetitions, …);
Dopo aver finito la ripetizione del mio suono lo stoppo, se non voglio riempire il canale con un altro suono:
PsychPortAudio('Stop', mychannel, 1, 1);
Infine chiudo il canale:
PsychPortAudio('Close', mychannel);
-**KbName ()** → per capire quale tasto ho premuto. Se scrivo un tasto e voglio il codice devo scriverlo tra apici, altrimenti se non lo scrivo tra apici lo considera come codice e mi dà il tasto corrispondente. Es. KbName('9') = 105 dove 105 è il codice di 9. KbName(105) = 9;
- **[x,y, button] = GetMouse()** → si attiva quando viene trovato nello script e presenta tre uscite: x,y e button; x e y sono le posizioni del mouse e button un vettore booleano con tanti elementi quanti sono i bottoni presenti nel mouse. Sono tutti 0 eccetto quello che, nel caso, viene premuto. Questo comando non prende le coordinate della finestra (rect) ma dello schermo;
-**[numclicks, x, y, button] = GetClicks** → si attiva quando premo sul bottone ed è costituito da 4 uscite: dal numero dei click sul bottone, dalla posizione del mouse (x,y) quando pigio il bottone e dal vettore booleano di elementi come per GetMouse. Anche questo comando non prende le coordinate della finestra ma dello schermo;
-**SetMouse(x,y)** → setta la posizione del mouse nel punto xy (= mi fa muovere il mouse in quel punto). Non prende le coordinate della finestra ma dello schermo;
-**HideCursor / ShowCursor** → mi permette di nascondere/mostrare temporaneamente il puntatore.

Remember:
-**RGB triplets** → la mia immagine sarà la sovrapposizione di 3 matrici di uguale dimensione, la prima rappresenta il grado di rosso, la seconda di verde e la terza di blu. La matrice è **tridimensionale** (ho (200x200)**x3**). Per accedere al 1° elemento della matrice: (1 1 1) dove l'ultimo 1 indica in quale matrice mi trovo. Se ho c = rgb2gray(A) ho una matrice solo 200x200, con valori che vanno da 0 a 255. Infatti ho una sola matrice quando ho un'immagine con soli toni di grigio;
-**Creazione immagine** → >>b = a(: , : , 1) → ho messo in b il primo strato (RED) della matrice a
>> image(b)
>> colormap(gray(256)) → per avere l'immagine in toni di grigio
>> b = rgb2gray(a)
-**Modifiche a un'immagine** → per renderla più luminosa sommo a tutti gli elementi dell'immagine A uno stesso numero, per renderla più scura lo sottraggo a tutti gli elementi;
es. A = floor(min(A+128,255)). Per ottenere il negativo dell'immagine: AN = 256-A, mentre per ruotarla uso A', poiché essendo una matrice valgono le stesse operazioni delle matrici;

-Le parentesi quadre nella sintassi di una funzione indicano che l'argomento non è obbligatorio;
-Posso uguagliare due vettori, non un carattere (o la sua posizione) e un vettore;
- [ ] per vettori e matrici, ' ' per caratteri o stringhe considerate caratteri, " " per stringhe considerate un unico elemento e ( ) se vogliamo le posizioni. La scrittura 'Mario' è equivalente a
b = char("Mario");
-Quando instanzio una variabile essa deve contenere dati omogenei: devono essere o tutti numeri, o tutti caratteri o tutte stringhe, non misti;
- 'Mario' = 5 caratteri, mentre "Mario" è una stringa;
-Per le procedure devo scrivere input e devo instanziare la x;
-Per stampare gli elementi di una matrice ho bisogno di 2 cicli for annidati, poi disp(x, nrows, ncols);
-Il nome della funzione deve essere lo stesso del file;
-Per avere il numero max/min di elementi di un vettore devo scorrerlo tutto; quindi per la matrice ho bisogno di 2 variabili di ciclo (una che scorra le righe e l'altra le colonne);
-Quando scrivo una funzione devo instanziare gli output con il ; per non stamparli;
- function [min, max] = minmax(x) → per richiamarla nel Command Windows:
[a, b] = minmax([2,4,5,6]) dove a = min, b = max e il vettore di numeri la x;
-Gestione dei file (lettura/scrittura);
-Caratteristiche di un'onda;
-Poiché i video sono sequenze di immagini in successione vanno creati tramite cicli (for);
-Come sono gestiti i rettangoli in Psychtoolbox e come mettere le cose in background;
-Fare lo script della reaction time (p. 56).