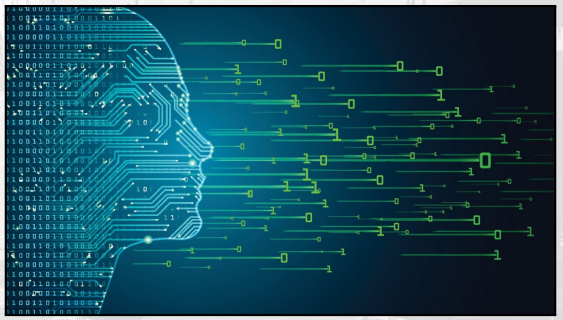
Programmazione Informatica
6. MATLAB, parte 1: introduzione e matrici

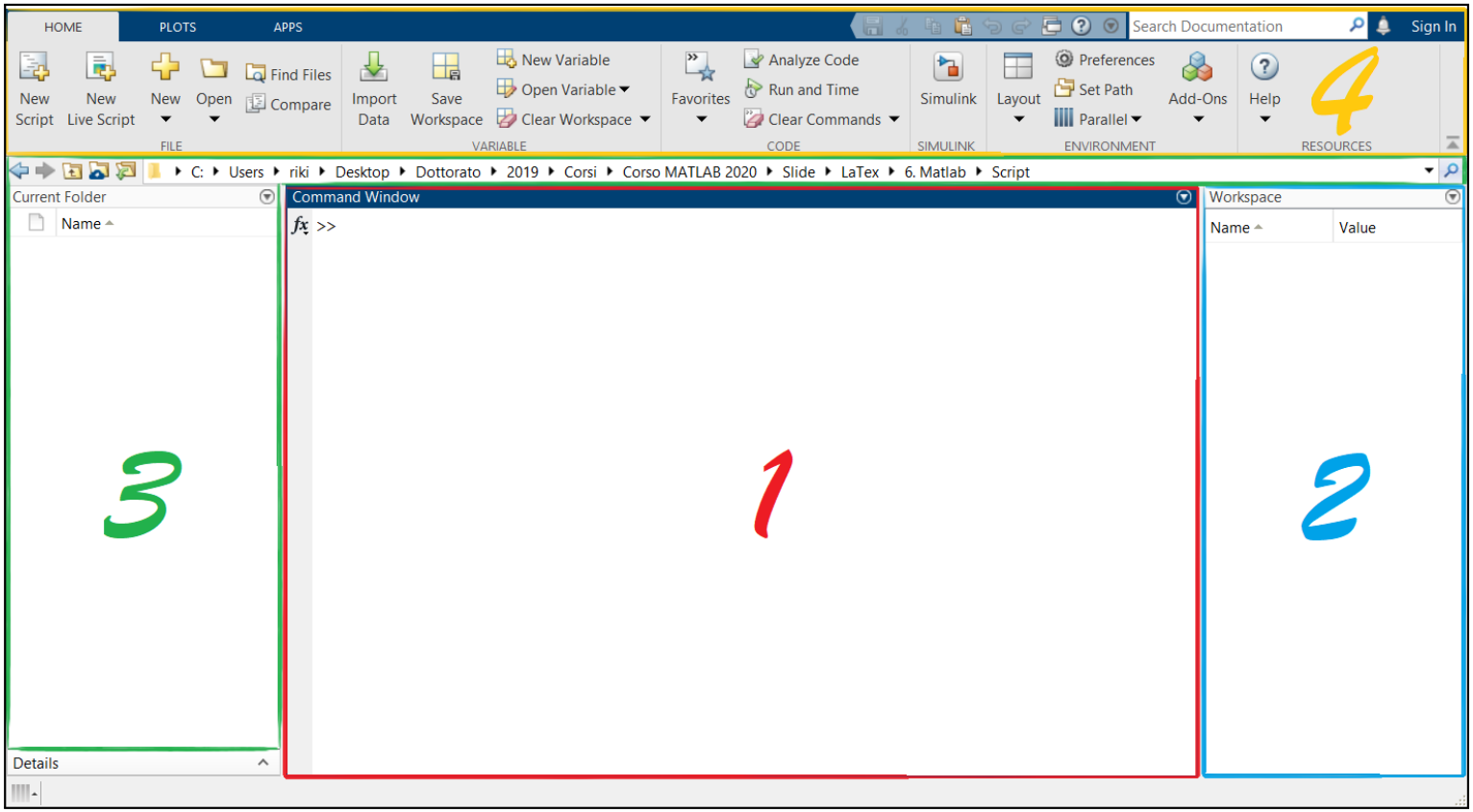
Riccardo Zamolo
rzamolo@units.it

Università degli Studi Trieste
Ingegneria Civile e Ambientale



A.A. 2023-24

La finestra di MATLAB appare così:



- 1 - Command Window: in questa sottofinestra, MATLAB può essere usato interattivamente inserendo comandi o espressioni dopo il simbolo di *prompt* “>>”;
- 2 - Workspace: contiene la lista ed il contenuto delle variabili utilizzate;
- 3 - Current Folder: percorso contenente i file di lavoro;
- 4 - Toolstrip: contiene strumenti e funzionalità varie, suddivisi in tab.

- Digitando `doc` al prompt e premendo invio (oppure andando in $HOME \rightarrow Help \rightarrow Documentation$) viene visualizzata la documentazione di MATLAB che riporta nel dettaglio il funzionamento di tutti i comandi utilizzabili, ordinati per argomento.
- Gli script, ossia i programmi veri e propri che l'interprete di MATLAB eseguirà, vanno tipicamente scritti nell'*Editor*, accessibile andando in $HOME \rightarrow New\ Script$. Dall'editor è possibile eseguire uno script andando in $EDITOR \rightarrow Run$.
- Ogni sottofinestra di MATLAB può essere trasformata in una finestra esterna (e viceversa) cliccando sul simbolo  in alto a destra nella finestra $\rightarrow Dock/Undock$.
- La configurazione delle finestre/sottofinestre può essere personalizzata andando in $HOME \rightarrow Layout$.

- Le variabili vengono definite mediante l'assegnazione ("←"):

  `nome_variabile = espressione`

```
>> x = 5
x =
     5
```

- I nomi delle variabili devono iniziare con una lettera; ci sono dei nomi che non è possibile usare perchè utilizzati per altre funzionalità (parole chiave, funzioni native).
- In MATLAB il dato contenuto nella variabile identifica il tipo della variabile. Tipi più usati:
  - numerici: **`double`** (default) e **`single`** in virgola mobile; **`int8`**, **`int16`**, **`int32`**, **`int64`** e le loro versioni senza segno **`uint8`**, **`uint16`**, **`uint32`**, **`uint64`** per gli interi;
  - logici (booleani): **`logical`**;
  - testo: **`char`** per singoli caratteri, **`string`** per stringhe ($\neq$ array di **`char`**).
- La conversione tra un tipo e l'altro (quando possibile) può essere fatta utilizzando il nome del tipo stesso.
  Esempio di conversione da **`double`** (default) a **`int32`**:

```
>> n = int32(10.8)
n =
     int32
     11
```

- Il comando `class` fornisce il tipo di una variabile o di una espressione:

```
>> class(10.8)
ans =
     'double'
```

dove `ans` è la varibile di default utilizzata per memorizzare le espressioni senza assegnazione.

- Operazioni scalari tra numeri: `+,-,*,/,^`:

```
>> y = ( (1.5+7) * 5.2 / 9 - 4) ^ 2
y =
     0.8301
```

- Con il comando `format short` vengono mostrate solo 4 cifre decimali (default) se il numero non è intero; `format long` permette di passare a 15 cifre decimali:

```
>> format long ;
>> y = ( (1.5+7) * 5.2 / 9 - 4) ^ 2
y =
     0.830123456790124
```

- Notazione esponenziale compatta in base 10:
  `(mantissa)e,E,d,D(esponente) = mantissa * 10^esponente`

```
>> x = 1.5e-5 ;
>> y = 999.999E99 ;
>> z = 6.022d23 ;
>> w = .5D-5
w =
     5.0000e-06
```

- Costanti: `pi` (π), `i,j` (unità immaginaria), `Inf` (infinito), `NaN` (Not a Number), `eps` (epsilon di macchina ϵ per le variabili `double`)

```
>> x = 1 + 2j
x =
     1.0000 + 2.0000i
>> x = 1 / 0
x =
     Inf
>> x = 0 / 0
x =
     NaN
```

```
>> eps % = eps(1)
ans =
     2.2204e-16
>> eps( single(1) )
ans =
     single
     1.1921e-07
```

- Gli scalari complessi sono di default dei `double`, con attributo `complex`. Utilizzando il comando `whos` si possono visualizzare tutte le informazioni di tutte le variabili utilizzate:

```
>> x = 1 ;
>> z = 1 + 2j ;
>> whos
     Name        Size                Bytes  Class     Attributes
     x           1x1                     8  double
     z           1x1                    16  double    complex
```

- Tutte le operazioni scalari `+,-,*,/,^` possono essere utilizzate con i numeri complessi:

```
>> ( 1 + 2j ) * ( -1.5 + 2.5j ) ;
ans =
     -6.5000 - 0.5000i
```

- Tutte le principali funzioni matematiche sono già implementate in MATLAB (*funzioni native* o *built-in functions*). La lista completa di queste funziponi può essere reperita nella documentazione sotto *MATLAB* → *Mathematics* → *Elementary Math*:
  - divisioni intere e arrotondamento: `mod`, `floor`, `ceil`, `round`, ecc.
  - trigonometriche: `sin`, `asin`, `cos`, `acos`, `tan`, `atan`, `atan2`, ecc.
  - iperboliche: `sinh`, `asinh`, `cosh`, `acosh`, `tanh`, `atanh`, ecc.
  - esponenziali: `exp`, `log`, `sqrt`, ecc.
  - numeri complessi: `abs`, `angle`, `conj`, `real`, `imag`, ecc.
  - combinatoria: `factorial`, `nchoosek`, ecc.
  - numeri casuali: `rand`, `randn`, ecc.

```
>> sin( pi )
ans =
     1.2246e-16
>> acos( -1 )
ans =
     3.1416
>> exp( 1 )
ans =
     2.7183
```

```
>> abs( 1 + j )
ans =
     1.4142
>> angle( -1 )
ans =
     3.1416
>> factorial( 5 )
ans =
     120
```

- Ad eccezione delle funzioni che operano su interi, tutte le altre funzione sono definite anche per argomenti complessi:

```
>> exp( pi * j )
ans =
     -1.0000 + 0.0000i
```

- Per le variabili Booleane si utilizza il tipo `logical`. I due possibili stati che può assumere una variabile di questo tipo possono essere definiti da `false` (=`logical(0)`) e `true` (=`logical(1)`):

```
>> false
ans =
     logical
     0
```

```
>> true
ans =
     logical
     1
```

- Una variabile Booleana può anche essere definita mediante un'*operazione relazionale* facendo uso degli operatori relazionali `>`,`<`,`>=`,`<=`,`==` (identità), `~=` (disuguaglianza):

```
>> x = 2.5 > 1
x =
     logical
     1
```

```
>> y = 5 == 5
y =
     logical
     1
```

- Le espressioni logiche (funzioni Booleane) sono definite dall'applicazione dei seguenti operatori logici alle variabili Booleane:
  - somma logica ($+$, OR): `x | y` oppure `or(x,y)`;
  - prodotto logico ($\cdot$, AND): `x & y` oppure `and(x,y)`;
  - complemento logico ($\bar{}$, NOT): `~x` oppure `not(x)`;

```
>> ( 1 > 0 ) & ( .5 == .5 )
ans =
     logical
     1
```

```
>> ~( 0 | 1 ) == ~0 & ~1
ans =
     logical
     1
```

- Calcolare le due soluzioni x_1 e x_2 dell'equazione di secondo grado

$$ax^2 + bx + c = 0$$

definendo 3 particolari valori per i coefficienti a, b e c. Si verifichi inoltre che i due valori calcolati siano effettivamente soluzione dell'equazione.

```
% Definizione coefficienti
a = 1 ;
b = 2 ;
c = 3 ;

% Calcolo discriminante
delta = sqrt( b^2 - 4*a*c ) ;

% Calcolo radici x1 e x2
x1 = ( -b + delta ) / ( 2*a )
x2 = ( -b - delta ) / ( 2*a )

% Verifica
p1 = a*x1^2 + b*x1 + c
p2 = a*x2^2 + b*x2 + c
```

- Dall'esecuzione del precedente script si ottiene:

```
x1 =
      -1.0000 + 1.4142i
x2 =
      -1.0000 - 1.4142i
p1 =
      -4.4409e-16
p2 =
      -4.4409e-16
```

- MATLAB è l'abbreviativo di MATrix LABoratory: è perciò espressamente concepito per operare su grandezze matriciali. L'uso delle matrici, con le relative operazioni, è l'elemento chiave di questo linguaggio.
- Ad esclusione di alcuni elementi particolari, ogni grandezza è trattata di default come una matrice. Ad esempio, qualsiasi variabile scalare è infatti trattata come una matrice 1×1, come si può immediatamente verificare attraverso il comando `size` che restituisce le dimensioni (numero di righe e di colonne) dell'argomento passato:

```
>> size(1)
ans =
     1     1
```

- In MATLAB una matrice $\mathbf{A}$ di dimensione $m \times n$ (array bidimensionale) è un insieme di elementi dello stesso tipo (valori numerici, logici o testo) organizzati secondo m righe e n colonne. Ogni elemento a_{ij} è identificato dai suoi indici i (riga) e j (colonna):

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

- Un vettore riga **r** è un caso speciale di matrice con una sola riga, quindi di dimensione $1 \times n$, un vettore colonna **c** è un caso speciale di matrice con una sola colonna, quindi di dimensione $n \times 1$. Nei vettori (array monodimensionali) è sufficiente un solo indice per identificare ogni elemento:

$$\mathbf{r} = \begin{bmatrix} r_1 & r_2 & \cdots & r_n \end{bmatrix} \qquad \mathbf{c} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix}$$

- È possibile utilizzare array con più di 2 dimensioni, per esempio nel caso di un'immagine a colori dove sono necessarie 3 dimensioni (2 dimensioni per l'indicizzazione spaziale dei pixel, 1 dimensione per l'indice del canale).
- Un vettore riga `r` si può definire esplicitamente a partire dagli elementi che lo compongono mediante la concatenazione in riga:

```
>> r = [ 4 7 1 -8 ]
r =
     4     7     1    -8
```

```
>> r = [ 4 , 7 , 1 , -8 ]
r =
     4     7     1    -8
```

- Nel precedente caso `r` sarà un vettore di tipo `double` in quanto definito a partire da 4 numeri che sono tutti assunti di tipo `double` di default.

- L'uso della concatenazione (in riga) è possibile non solo a partire dai valori scalari ma anche su vettori (riga) a loro volta:

```
>> a = [ 1 2 ] ;
>> b = [ 4 , 5 ] ;
>> c = [ 7.5 , 8.9 ] ;
>> [ a , b c ]
ans =
      1.0000    2.0000    4.0000    5.0000    7.5000    8.9000
```

- Un vettore colonna `c` si può definire esplicitamente a partire dagli elementi che lo compongono mediante la concatenazione in colonna. Anche in questo caso la concatenazione (in colonna) può avvenire su vettori (colonna) a loro volta:

```
>> c = [ 4 ; 7 ; -8 ]
c =
     4
     7
    -8
```

```
>> a = [ 1 ; 2 ] ;
>> b = [ 3 ; 4 ; 5.5 ] ;
>> [ a ; b ]
ans =
     1.0000
     2.0000
     3.0000
     4.0000
     5.5000
```

- Quando è necessario creare un vettore (riga) di numeri che vanno da `v1` a `v2` con passo costante `step` noto, si utilizza la notazione ":" (*colon*):

  `v1:step:v2`

  Non è necessario nè che `v2>v1` nè che `step` sia una frazione intera di `v2-v1`, ma devono avere segno concorde:

```
>> r = 1 : .5 : 2.2
r =
    1.0000    1.5000    2.0000
```

```
>> r = 10 : -1 : 8
r =
    10     9     8
```

- Quando il passo è unitario (`step=1`), si utilizza la notazione compatta `v1:v2`

```
>> r = .5 : 6
r =
    0.5000    1.5000    2.5000    3.5000    4.5000    5.5000
```

- Nel caso sia invece necessario ottenere il vettore (riga) di `n` numeri che vanno esattamente da `v1` a `v2` con spaziatura uniforme, si utilizza la seguente funzione:

  `linspace(v1,v2,n)`

```
>> r = linspace(0,1,6)
r =
    0    0.2000    0.4000    0.6000    0.8000    1.0000
```

- Combinando le concatenazioni in riga ed in colonna, è possibile definire esplicitamente una matrice:

```
>> A = [ 0 1 2 ; 3 5 8 ; 5 0 5 ]
A =
     0     1     2
     3     5     8
     5     0     5
```

```
>> A = [ 1 0 ; 0 1 ] ;
>> B = [ 1 2 ; 3 4 ] ;
>> C = [ 0 0 ; 0 0 ] ;
>> D = [ 5 0 ; 0 5 ] ;
>> [ A B ; C D ]
ans =
     1     0     1     2
     0     1     3     4
     0     0     5     0
     0     0     0     5
```

- L'accesso in lettura o in assegnazione agli elementi di una matrice `A` avviene mediante l'uso di indici (`i,j,k` interi):
  - `A(i,j)`: elemento a_{ij} di una matrice;
  - `A(i)` oppure `A(1,i)`: elemento a_i di un vettore riga;
  - `A(i)` oppure `A(i,1)`: elemento a_i di un vettore colonna;
  - `A(k)`: elemento a_{ij} di una matrice mediante indice lineare k calcolato per colonne *(column-major)*:

$$\texttt{A} = \begin{bmatrix} \texttt{A(1,1)} & \texttt{A(1,2)} & \cdots & \texttt{A(1,n)} \\ \texttt{A(2,1)} & \texttt{A(2,2)} & \cdots & \texttt{A(2,n)} \\ \vdots & \vdots & \ddots & \vdots \\ \texttt{A(m,1)} & \texttt{A(m,2)} & \cdots & \texttt{A(m,n)} \end{bmatrix} = \begin{bmatrix} \texttt{A(1)} & \texttt{A(m+1)} & \cdots & . \\ \texttt{A(2)} & \texttt{A(m+2)} & \cdots & . \\ \downarrow & \downarrow & \downarrow & \downarrow \\ \texttt{A(m)} & \texttt{A(2m)} & \cdots & \texttt{A(m*n)} \end{bmatrix}$$

- Accesso in lettura o assegnazione:

```
>> A = [ 1 4 7 ; ...
         2 5 8 ; ...
         3 6 9 ] ;
>> A(3,3)
ans =
      9

>> A(9)
ans =
      9
```

```
>> A(3,3) = 100
A =
     1     4     7
     2     5     8
     3     6   100

>> A(9) = -1
A =
     1     4     7
     2     5     8
     3     6    -1
```

- Gli indici usati per l'accesso possono essere a loro volta dei vettori per effettuare estrazioni o assegnazioni vettoriali/matriciali (`vi`,`vj`,`vk` vettori di interi):
  - `A(vi,vj)`: sottomatrice di `A` individuata dall'intersezione delle righe `vi` con le colonne `vj`;
  - `A(vi)` oppure `A(1,vi)`: vettore riga degli elementi di `A`, a sua volta vettore riga, con indice `vi`;
  - `A(vi)` oppure `A(vi,1)`: vettore colonna degli elementi di `A`, a sua volta vettore colonna, con indice `vi`;
  - `A(vk)`: vettore degli elementi della matrice `A` individuati dagli indici lineari `vk`. `A(vk)` è vettore colonna se `vk` è vettore colonna, riga in caso contrario.

- Lettura o assegnazione matriciale con due indici:

```
>> A = [ 0 1 2 ; ...
         3 5 8 ; ...
         9 0 9 ] ;
>> vi = 1:2 ; % oppure [1 2]
>> vj = 2:3 ; % oppure [2 3]
>> A(vi,vj)
ans =
     1     2
     5     8
```

```
>> A = [ 0 1 2 ; ...
         3 5 8 ; ...
         9 0 9 ] ;
>> vi = 1:2 ;
>> vj = 2:3 ;
>> A(vi,vj) = [ -2 -3 ; ...
                 -4 -5 ]
A =
     0    -2    -3
     3    -4    -5
     9     0     9
```

- Lettura o assegnazione vettoriale con un indice lineare:

```
>> A = [ 0 1 2 ; ...
         3 5 8 ; ...
         9 0 9 ] ;
>> vk = 1:5 ;
>> A(vk)
ans =
     0     3     9     1     5
```

```
>> A = [ 0 1 2 ; ...
         3 5 8 ; ...
         9 0 9 ] ;
>> vk = 1:5 ;
>> A(vk) = 11:15
A =
    11    14     2
    12    15     8
    13     0     9
```

- Indici particolari: **end** (ultimo valore per quell'indice) e ":" (colon, tutti i valori per quell'indice)

```
>> A = [ 1 2 ; 3 4 ] ;
>> A(end,:)
ans =
     3     4
```

- In MATLAB l'operatore di trasposizione (coniugata) è l'apice `'` :

```
>> v = [ 1 ; 2 ; 3 ] ; % colonna
>> v'
ans =
     1     2     3
```

```
>> A = [ 0 1 2 ; ...
         3 5 8 ] ;
>> A'
ans =
     0     3
     1     5
     2     8
```

- Data una matrice `A` di dimensione `m` $\times$ `n`, `A(:)` fornisce quindi il vettore colonna di tutti gli elementi di `A` presi per colonne. L'operazione inversa può essere eseguita mediante la funzione `reshape`:

  `reshape( A_in , m , n )`

  che fornisce una matrice `m` $\times$ `n` riempita con tutti gli elementi della matrice `A_in`, seguendo sempre l'ordine per colonne. Ovviamente il numero totale degli elementi di `A_in` deve essere `m` $\cdot$ `n`:

```
>> v = [ 1 2 3 4 ] ;
>> reshape( v , 2 , 2 )
ans =
     1     3
     2     4
```

```
>> A = [ 1 3 ; ...
         2 4 ] ;
>> reshape( A , 4 , 1 ) % = A(:)
ans =
     1
     2
     3
     4
```

- Alcune funzioni utili: `size(A)` restituisce le dimensioni di `A` (numero di righe e di colonne) sotto forma di vettore; `size(A,dim)` restituisce il numero di righe (`dim=1`) o di colonne (`dim=2`); `length(A)` restituisce la massima delle dimensioni di una matrice `A`, corrispondente quindi alla lunghezza nel caso di un vettore riga/colonna:

```
>> A = reshape( 1:6 , 2 , 3 ) ;
>> size_A = size( A )
size_A =
     2     3
```

```
>> v = linspace( 0 , 1 , 1000 ) ;
>> length( v )
ans =
        1000
```

- Si possono inizializzare matrici nulle e matrici identità, di dimensione $\mathtt{m} \times \mathtt{n}$ e di tipo `type`, utilizzando rispettivamente:

`zeros( m , n , type )` `eye( m , n , type )`

```
>> Z = zeros( 3 , 2 , 'int32' )
Z =
  3×2 int32 matrix
   0   0
   0   0
   0   0

>> Z = zeros( 3 , 2 ) % double
Z =
     0     0
     0     0
     0     0
```

```
>> I = eye( 3 , 3 , 'int32' )
I =
  3×3 int32 matrix
   1   0   0
   0   1   0
   0   0   1

>> I = eye( 3 , 3 ) % double
I =
     1     0     0
     0     1     0
     0     0     1
```

- Creare una matrice quadrata **A** di dimensione $n \times n$ che soddisfi la proprietà $a_{ij} = i$ se $i = j$ (diagonale principale), $a_{ij} = 0$ altrimenti.
- Sarà sufficiente utilizzare la funzione **zeros** per allocare una matrice nulla, utilizzando poi l'indicizzazione lineare per assegnare i valori sulla diagonale:

```
% Dimensione
n = 5 ;

% Allocazione con matrice nulla
A = zeros( n , n ) ;

% Vettore dei valori diagonali
vi = 1 : n ;

% Vettore degli indici lineari degli elementi diagonali
vk = 1 : (n+1) : n^2 ;

% Assegnazione vettoriale dei termini diagonali
A(vk) = vi
```

- Dall'esecuzione del precedente script si ottiene:

```
A =
     1     0     0     0     0
     0     2     0     0     0
     0     0     3     0     0
     0     0     0     4     0
     0     0     0     0     5
```

- MATLAB è nato per operare su matrici, perciò le operazioni elementari `+,-,*,^` sono nativamente definite in senso matriciale/algebrico:
  - `A+B,A-B`: somma $\mathbf{A}+\mathbf{B}$, $a_{ij}+b_{ij}$, differenza $\mathbf{A}-\mathbf{B}$, $a_{ij}-b_{ij}$, tra due matrici della stessa dimensione (elemento per elemento);
  - `u*A`: prodotto scalare×matrice $u\mathbf{A}$, $u \cdot a_{ij}$;
  - `A*B`: prodotto matriciale $\mathbf{AB}$ (righe per colonne);
  - `A^n`: prodotto matriciale di una matrice quadrata per se stessa n volte $\underbrace{\mathbf{A}\mathbf{A}\cdots\mathbf{A}}_{n \text{ volte}}$.

```
>> A = [ 1 2 ; ...
         3 4 ] ;
>> 3 * A
ans =
     3     6
     9    12

>> v = [ 1 ; 1.5 ] ; % colonna
>> A * v
ans =
     4
     9

>> v' * A
ans =
    5.5000    8.0000
```

```
>> v' * v % Prodotto scalare
ans =
    3.2500

>> A = [ 1 1 ; ...
         0 1 ] ;
>> A ^ 15
ans =
     1    15
     0     1

>> ( A - eye(2,2) ) ^ 2
ans =
     0     0
     0     0
```

- Sono possibili altre operazioni elementari, in forma compatta, non direttamente definte in senso algebrico:
  - `u+A`: somma scalare+matrice, $u + a_{ij}$, (ad ogni elemento di `A` viene sommato `u`);
  - `r+A`: somma vettore riga+matrice, ad ogni riga di `A` viene sommato il vettore riga `r`;
  - `c+A`: somma vettore colonna+matrice, ad ogni colonna di `A` viene sommato il vettore colonna `c`;
  - `(A .* B),(A ./ B),(A .^ B)`: operazioni elemento per elemento (*element-wise*). Come per la somma, `A` e `B` possono essere scalari, vettori riga/colonna o matrici, di dimensione opportuna.

```
>> A = [ 1 2 ; ...
         3 4 ] ;
>> 5 + A
ans =
     6     7
     8     9

>> r = [ 1 10 ] ;
>> r + A
ans =
     2    12
     4    14

>> c = [ 1 ; 10 ] ;
>> c + A
ans =
     2     3
    13    14
```

```
>> B = [ 0 1 ; ...
         2 0 ]
>> A .* B
ans =
     0     2
     6     0

>> A ./ B
ans =
       Inf    2.0000
    1.5000       Inf

>> A .^ B
ans =
     1     2
     9     1
```

- Dato il vettore riga `a`$=(a_0, a_1, \ldots, a_n)$ dei coefficienti del polinomio

$$p(x) = a_0 + a_1 x + \cdots + a_n x^n$$

si calcoli il polinomio per $x = \pi$.

- Bisognerà innanzitutto determinare n a partire dal vettore `a` dato, utilizzando poi la forma

$$p(x) = \underbrace{(a_0, a_1, \ldots, a_n)}_{\texttt{a}} \times \underbrace{(x^0, x^1, \ldots, x^n)}_{\texttt{potenze\_x}}{}^T = \texttt{a * potenze\_x'}$$

per calcolare il polinomio attraverso un prodotto scalare.

```
% x vale pi greco
x = pi ;

% Grado del polinomio
n = length( a ) - 1 ;

% Vettore riga degli esponenti
e = 0 : n ;

% Vettore riga delle potenze di x (scalare .^ vettore, elemento per elem.)
potenze_x = x .^ e ;

% Calcolo del polinomio attraverso prodotto scalare
p = a * potenze_x'
```

- Dato il vettore riga `a`$=(a_0, a_1, \ldots, a_n)$ dei coefficienti del polinomio

$$p(x) = a_0 + a_1 x + \cdots + a_n x^n$$

si calcoli il polinomio su $N = 1000$ valori di x equidistanziati nell'intervallo $[0, \pi]$.

- Utilizzeremo sempre la forma matriciale riga-colonna:

$$\mathbf{p(x)} = \underbrace{(a_0, a_1, \ldots, a_n)}_{\texttt{a}} \times \underbrace{\begin{pmatrix} x_1^0, x_2^0, \ldots, x_N^0 \\ x_1^1, x_2^1, \ldots, x_N^1 \\ \vdots \quad \vdots \quad \cdots \quad \vdots \\ x_1^n, x_2^n, \ldots, x_N^n \end{pmatrix}}_{\texttt{potenze\_x}} = \texttt{a * potenze\_x}$$

```
% Vettore riga dei N=1000 valori equidistanziati di x su [0,pi]
x = linspace( 0 , pi , 1000 ) ;

% Grado del polinomio
n = length( a ) - 1 ;

% Vettore colonna degli esponenti
e = (0 : n)' ;

% Matrice delle potenze di x (vettore riga .^ vettore colonna)
potenze_x = x .^ e ;

% Calcolo del polinomio attraverso prodotto matriciale
p = a * potenze_x ;
```

- Essendo MATLAB nato per operare su matrici/vettori, le funzioni elementari sono nativamente definite per operare su argomenti matriciali/-vettoriali, comprese le funzioni per la conversione di tipo: la funzione viene valutata elemento per elemento:

```
>> x = linspace( 0 , pi/2 , 7 ) ; % Vettore riga 1x7
>> y = sin( x )
y =
    0    0.2588    0.5000    0.7071    0.8660    0.9659    1.0000
>> int32( y )
ans =
  1×7 int32 row vector
   0   0   0   1   1   1   1
```

- Vi sono poi alcune utili funzioni pensate per operare su grandezze tipicamente matriciali/vettoriali:
  - **sum, cumsum, prod, cumprod**: somma e prodotto totale/cumulativo di tutti gli elementi di una matrice, per riga o per colonna;
  - **min, max**: minimo e massimo (ed eventualmente il relativo indice) degli elementi di una matrice, per riga o per colonna.

```
>> v = [ 1 3 -2 7 0 ] ;
>> sum( v )
ans =
     9
>> max( v )
ans =
     7
```

```
>> A = [  1  2 ; ...
         10 20 ] ;
>> sum( A , 1 )
ans =
     11    22
>> sum( A , 2 )
ans =
     3
     30
```

- Calcolare la somma delle seguenti potenze dei numeri naturali:

$$P_n = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$

$$Q_n = 1^2 + 2^2 + 3^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}$$

e verificare che le somme siano corrette attraverso le formule indicate per $n = 10$.

```
% Scelta di un valore particolare per n
n = 10 ;

% Vettore riga degli interi da 1 a n
i = 1 : n ;

% Pn
Pn          = sum( i )
Pn_formula = n * ( n + 1 ) / 2

% Qn
Qn          = sum( i .^ 2 )
Qn_formula = n * ( n + 1 ) * ( 2*n + 1 ) / 6
```

- Dall'esecuzione del precedente script si ottiene:

```
Pn =
     55

Pn_formula =
     55
```

```
Qn =
  385

Qn_formula =
  385
```

- Matrici/vettori Booleani (ossia di tipo `logical`) sono definibili usando in maniera matriciale/vettoriale gli operatori relazionali `>`,`<`,`>=`,`<=`,`==`,`~=`:

```
>> a = [ 1 2 3 4 5 ] ;
>> b = [ 5 4 3 2 1 ] ;
>> v_booleano = a >= b
v_booleano =
  1×5 logical array
  0   0   1   1   1
```

```
>> a = [ 1 2 3 4 5 ] ;
>> v_booleano = a == 3
v_booleano =
  1×5 logical array
  0   0   1   0   0
```

- Sono ovviamente possibili le espressioni logiche matriciali/vettoriali utilizzando in maniera matriciale/vettoriale gli operatori logici:

```
>> x = logical( [ 0 0 1 1 ] ) ;
>> y = logical( [ 0 1 0 1 ] ) ;
>> x & y
ans =
  1×4 logical array
  0   0   0   1
>> ~( x | y )
ans =
  1×4 logical array
  1   0   0   0
>> ~x & ~y
ans =
  1×4 logical array
  1   0   0   0
```

- È possibile utilizzare matrici/vettori Booleani per indicizzare, in lettura o assegnazione, altre matrici/vettori con lo stesso numero di elementi per quell'indice:

```
>> v = [ 1 9 2 4 7 ] ;
% Ottenere da v i numeri > 3
>> ix = v > 3
ix =
  1×5 logical array
  0   1   0   1   1
>> v( ix )
ans =
     9     4     7
```

```
>> A = reshape( 1:16 , 4 , 4 )
A =
     1     5     9    13
     2     6    10    14
     3     7    11    15
     4     8    12    16
% Ottenere prima e ultima riga di A
>> ix = logical( [ 1 0 0 1 ] ) ;
>> A( ix , : ) % = A( [1 4] , : )
ans =
     1     5     9    13
     4     8    12    16
```

- Funzioni che operano su matrici/vettori Booleani:
  - **any**: OR applicato a tutti gli elementi di una riga, di una colonna o sulla matrice intera;
  - **all**: AND applicato a tutti gli elementi di una riga, di una colonna o sulla matrice intera;
  - **find**: restituisce gli indici degli elementi **true** di una matrice; l'indice può essere lineare oppure doppio, riga e colonna.

```
>> any( 10:15 > 13 )
ans =
  logical
  1
```

```
>> find( [10 11 12 13 14 15] > 13 )
ans =
     4     5
```

- Determinare un'approssimazione dei 3 zeri della funzione

$$f(x) = \sin(2x) - x$$

nell'intervallo $[-\pi/2, \pi/2]$. Si utilizzi la funzione `sign(x)` che resituisce ± 1 a seconda del segno di `x` (e 0 se `x=0`).

- Basterà calcolare la funzione in maniera vettoriale su un numero sufficientemente elevato di valori di x e cercare dove la funzione cambia segno:

```
% Suddividiamo l'intervallo dato in 1000 punti (per esempio)
x = linspace( -pi/2 , pi/2 , 1000 ) ;

% Calcolo funzione
f = sin(2*x) - x ;

% Segno della funzione
segno_f = sign(f) ;

% Cerchiamo dove il segno della funzione cambia
ix = find( segno_f(1:end-1) ~= segno_f(2:end) ) ;

% Approssimazione degli zeri (media di due punti adiacenti)
zeri = ( x(ix) + x(ix+1) ) / 2
```

- Dall'esecuzione del precedente script si ottiene (utilizzando `format long`):

```
zeri =
-0.946565954685213                        0    0.946565954685213
```

ed i valori esatto sono $x = \pm 0.947747133516990$, oltre ad $x = 0$.