Funzioni Built-in, di ordine superiore e Plot in Matlab

Informatica B

Francesco Trovò

francesco1.trovo@polimi.it

23 Novembre 2021

Esempi

Definire una funzione **`samplePolynomial`** che dato in ingresso

- un vettore di coefficienti **`polyCoeff`**
- un vettore **`interval`** che definisce un intervallo [a, b]

restituisce due vettori di 100 punti **`xx`** ed **`yy`** contenenti i punti che stanno sulla curva

$$y = C(1)x^{n-1} + C(2)x^{n-2} + \ldots + C(n-1)x^1 + C(n)$$

e le cui ascisse stanno in [a, b]

Soluzione

```
function [xx, yy] =
samplePolynomial(polyCoeff, interval)
% per essere certi che a <= b
a = min(interval);
b = max(interval);

xx = [a   : (b-a) / 100 : b];
% oppure xx = linspace(a , b, 100)
yy  = zeros(size(xx));

for ii = 1 :  1 : length(polyCoeff)
    yy = yy + polyCoeff(ii) *
xx.^(length(polyCoeff) - ii);
end
```

Funzioni Built-in

Alcune Funzioni Built-in per Gestire Array

Funzione	Significato
`zeros (n)`	Restituisce una matrice n×n di zeri
`zeros (m,n)`	Restituisce una matrice m×n di zeri
`zeros(size(arr))`	Restituisce una matrice di zeri delle dimensioni di `arr`
`ones(n)`	Restituisce una matrice n×n di uno
`ones(m,n)`	Restituisce una matrice m×n di uno
`ones(size(arr))`	Restituisce una matrice di uno della stessa dimensione di arr
`eye(n)`	Restituisce la matrice identità n×n
`eye(m,n)`	Restituisce la matrice identità m×n
`length(arr)`	Ritorna la dimensione più lunga del vettore
`size(arr)`	Ritorna un vettore [r c] con il numero r di righe e c di colonne della matrice; se arr ha più dimensioni ritorna array con numero elementi per ogni dimensione

Esempi

- `a = zeros(2);` ⟶ $\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$
- `b = ones(2,3);` ⟶ $\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$
- ```
  c = [1 2; 3 4];
  d = zeros(size(c));
  ```
  ⟶ $\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$
- `e = 2 * eye(2)` ⟶ $\begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$
  ```

Funzioni Aritmetiche

Funzione	Scopo
`ceil(x)`	approssima x all'intero immediatamente maggiore
`floor(x)`	approssima x all'intero immediatamente minore
`fix(x)`	approssima x all'intero più vicino verso lo zero
`[m, pos] = max(x)`	se x è un vettore, restituisce il valore massimo in x e, opzionalmente, la collocazione di questo valore in x; se x è matrice, restituisce il vettore dei massimi delle sue colonne
`[m, pos] = min(x)`	se x è un vettore, restituisce il valore minimo nel vettore x e, opzionalmente, la collocazione di questo valore nel vettore; se x è matrice, restituisce il vettore dei minimi delle sue colonne
`mean(x)`	se x è un vettore restituisce uno scalare uguale alla media dei valori di x; se x è una matrice, le medie dei vettori colonna di x
`mod(m, n)`	mod(m,n) è m – q .* n dove q = floor(m ./ n) se n ~= 0
`round(x)`	approssima x all'intero più vicino
`rand(n)`	restituisce una matrice di nxn numeri casuali con distribuzione uniforme tra 0,1

Funzioni `min` (e `max`) Applicate a Vettori e Matrici

```
>> b = [4 7 2 6 5]
b =  4     7     2     6
>> min(b)
ans = 2
>> [x y] = min(b)
x =  2
y =  3
>>
```

Con un solo output dà il valore del minimo

Con due output dà anche la posizione del minimo

```
>> a = [24 28 21; 32 25 27; 30 33 31; 22 29 26]
a = 24    28    21
    32    25    27
    30    33    31
    22    29    26
>> min(a)
ans = 22    25    21

>> [x y] = min(a)
x =  22    25    21
y =  4     2     1
```

Per una matrice dà vettore dei minimi nelle colonne

Per una matrice, con due risultati dà due vettori dei valori minimi nelle colonne e della loro posizione (riga)

Funzioni Aritmetiche

`sum(vettore)` calcola la somma degli elementi di **`vettore`**

`prod(vettore)` calcola il prodotto degli elementi di **`vettore`**

Es.

Funzione per il calcolo del fattoriale

```
function k = fattoriale2(n)
     k = prod([n : -1 : 1]);
```

Altre Funzioni Importanti

Calcolo delle dimensione degli array

- **`length(v)`** restituisce la lunghezza del vettore
- **`size(A)`** restituisce un vettore contenente le dimensioni dell'array **`A`** (come si vedono da whos)
- **`size(A, dim)`** restituisce il numero di elementi di **`A`** lungo la dimensione **`dim`**

NB: length su matrici restituisce la dimensione avente più elementi, ovvero

```
length(A) == max(size(A))
```

Funzioni Logiche Built in

Funzioni Logiche

Nome	Elemento Restituito
`all(x)`	Se x è un vettore, restituisce 1 se tutti gli elementi sono non-nulli, 0 altrimenti. Se x è una matrice, un vettore riga con lo stesso controllo colonna per colonna
`any(x)`	Se x è un vettore, restituisce 1 se esiste almeno un elemento non-nullo, 0 altrimenti. Se x è una matrice, un vettore riga con lo stesso controllo colonna per colonna
`isinf(x)`	Un vettore con 1 dove gli elementi di x sono `Inf`, 0 altrimenti
`isempty(x)`	1 se x è vuoto, 0 altrimenti
`isnan(x)`	Un vettore con 1 dove gli elementi di x sono `NaN`, 0 altrimenti
`finite(x)`	Un vettore con 1 dove gli elementi di x sono finiti, 0 altrimenti
`ischar(x)`	1 se x è di tipo char, 0 altrimenti
`isnumeric(x)`	1 se x è di tipo double, 0 altrimenti
`isreal(x)`	1 se x ha solo elementi con parte immaginaria nulla, 0 altrimenti

Altre Funzioni Logiche: `find`

`indx = find(x)` restituisce gli indici degli elementi **`true`** dell'array logical **`x`**

```
>> a = [5 6 7 2 10];
>> b = a > 5; % b =[0 1 1 0 1]
>> find(b)
   ans = 2 3 5
```

Versione breve:

```
>> a = [5 6 7 2 10]
>> find(a > 5)
   ans = 2 3 5
```

NB: la sintassi NON è **`find(expr)`** ma **`find(logical)`**

Altre Funzioni Logiche: `find`

- **`find`** restituisce gli indici e non estrae un valori (come invece posso fare utilizzando vettori di interi o vettori logici come indici di un vettore)

```
x = [5, -3, 0, 0, 8];
y = [2, 4, 0, 5, 7];
vals = y((x>0) & (y>0)) ->  vals = [2 7]
idx = find((x>0) & (y>0))-> idx  = [1 5]
```

- L'estrazione di un sottovettore con le operazioni logiche è più rapida dell'utilizzo di find

```
k = x(find(x>0))   %lento
k = x(x>0)         %veloce
```

Esempio

Scrivere una funzione **cerca** che controlla se un elemento **x** appartiene ad un vettore **vett** e, in caso affermativo, ne restituisce la posizione

```
function [pres, pos] = cerca(x, vett)
    p = 0;
    pos = [];
    for ii = 1:length(vett)
        if vett(ii) == x
            p = p + 1;
            pos(p) = ii;
        end
    end
    pres = p > 0;
```

Utilizzo in uno Script

```
>> A = [1, 2, 3, 4, 3, 4, 5, 4, 5, 6]
     A = 1 2 3 4 3 4 5 4 5 6
>> [p, idx] = cerca(4, A)
     p = 1
     idx = 4 6 8
```

Parametri formali di **`cerca`**: **`x, vet`**

Parametri attuali di **`cerca`**: **`4, A`**

Esempio

Scrivere una funzione **cerca** che controlla se un elemento **x** appartiene ad un vettore **vett** e, in caso affermativo, ne restituisce la posizione

```
function [pres, pos] = cerca2(x, v)
pres = 1;
pos = find(v == x);
if isempty(pos)
    pres = 0;
end
```

```
function [pres, pos] = cerca3(x, v)
pres = any(v == x);
pos = find(v == x);
```

Esercizio

Scrivere una funzione **closestVal** che prende in ingresso un vettore **vett** ed uno scalare **x** e restituisce il valore di **vett** più vicino ad **x**

```
function [closest, pos_closest] = ...
      closestVal(vett, val)
diff = vett - val;
abs_diff = abs(diff);
[~ , pos_closest] = min(abs_diff);
closest = vett(pos_closest);
closest = unique(closest); % returns unique
values in a vector
```

Esempio Importante

Scrivere una funzione che prende un vettore e rimuove tutti i valori uguali a 7 e invocarla su `v = [12, 4, 7, 14]`

Devo sovrascrivere

```
function vettore = rimuovi7(vettore)
    vettore(vettore == 7) = [];
```

```
>> v = [12, 4, 7, 14]
>> disp(v)
>> v = rimuovi7(v);
>> disp(v)
```

Non c'è modo diretto di modificare una variabile nel workspace principale all'interno del corpo di una funzione

Homework

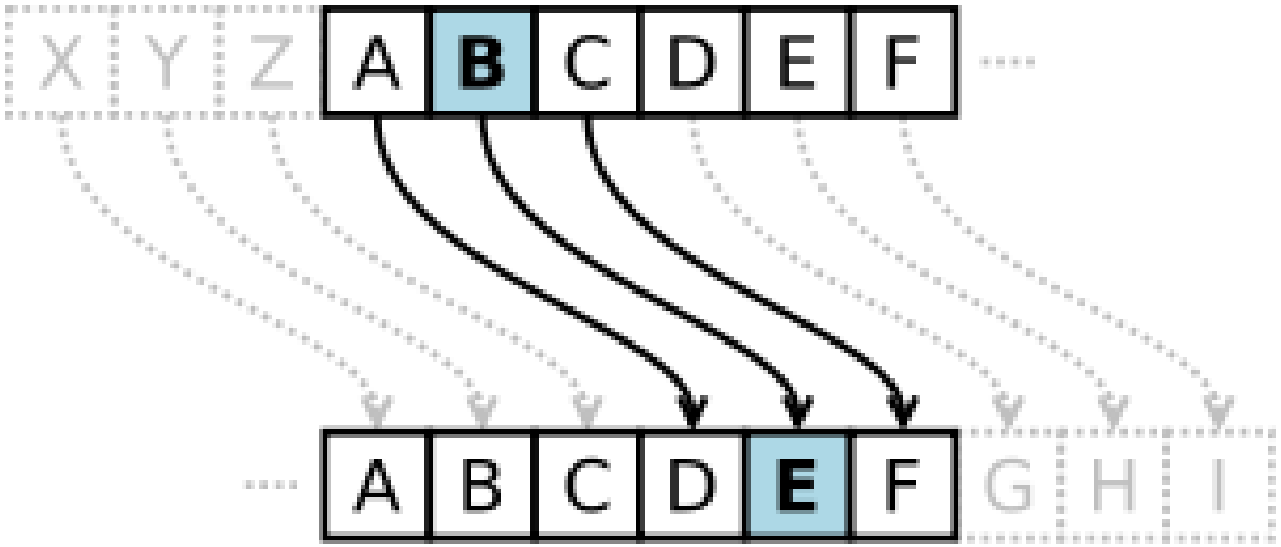
Scrivere un programma che richiede in ingresso due parole e determina se l'una è l'anagramma dell'altra:

- Uno script si occupa dell'acquisizione delle parole
- Implementare una funzione per creare l'istogramma delle parole
- Eseguire nello script il confronto tra istogrammi e visualizzare a schermo il risultato

Homework

Scrivere un programma che esegue la codifica di un testo utilizzando il cifrario di Cesare

Assicurarsi che la funzione sia in grado di eseguire anche la decodifica



Funzioni per Stringhe e Return

Funzioni per Stringhe

Esiste la funzione di **confronto**

```
comp = strcmp(str1, str2)
```

- `str1, str2` stringhe da confrontare
- `comp` valore booleano, vero se le due stringhe sono uguali e falso se sono differenti

`strcmpi(str1, str2)` funziona nello stesso modo, ma non fa differenze tra maiuscole e minuscole

NB: è diverso dal C dove in caso di stringhe uguali restituisce false

NB: in linea di principio è possibile confrontare le stringhe come due vettori, con l'operatore `==`, ma richiede che le **due stringhe abbiano le stesse dimensioni**

== vs. strcmp()

```
if('cane' == 'canguro')
     disp('uguali')
else
    disp('diverse')
End
>> Error using  ==
Matrix dimensions must agree.

if strcmp('cane', 'canguro')
    disp('uguali')
else
    disp('diverse')
end
>> diverse
```

Funzioni per Stringhe

Non occorre **`strlen`** -> **`length`** o **`size`**

Non occorre **`strcpy`** -> come la copia di array

Esiste la funzione di **ricerca**

```
idx = strfind(text, pattern)
```

- **`pattern`** stringa da ricercare
- **`text`** stringa in cui ricercare
- **`idx`** vettore contenente gli indici di tutte le occorrenze, vuoto se non ce ne sono

Comando Return

Può essere usato per terminare l'esecuzione di una funzione

```
function [p, m] = cercaMultiplo(v, a)
for k = 1:length(a)
    if mod(a(k), v) == 0
        p = k;
        m = a(k);
        return;  %restituisce il primo
multiplo incontrato
    end;
end;
p = 0; m = 0;  %eseguite solo se non trovato
alcun multiplo
```

Homework

Implementare la propria versione delle funzioni

- **`strcmpr`**
- **`any`**
- **`all`**

Scrivere una funzione **`replacePattern`** per sostituire, all'interno di una stringa **`txt`**, tutte le occorrenze di una determinata stringa **`s1`** con **`s2`**

Si assuma inizialmente che **`s1`** e **`s2`** abbiano la stessa lunghezza

Se ne scriva una versione che senza l'assunzione di ugual lunghezza

NB: `replacePattern` è implementata in Matlab dalla funzione **`strrep`**

Funzioni I/O

Operazioni di Input

- La funzione **`input`** apre una finestra di dialogo con l'utente e permette di inserire generiche istruzioni Matlab

```
a = input(txtToShow)
```

visualizza **`txtToShow`** nella command window, attende una generica istruzione Matlab, il cui valore viene assegnato ad **`a`**

```
a = input(txtToShow, 's')
```

visualizza **`txtToShow`** nella command window, attende una stringa, che viene assegnata ad **`a`**

- La funzione **`num2str(A)`** trasforma la matrice **`A`** in ingresso in una rappresentazione di stringa

Operazioni di Output

- I risultati di un'operazione sono mostrati immediatamente se non si inserisce il `;`
- **`disp()`** accetta come parametro un array e lo stampa a schermo
- **`fprintf()`** accetta come paramtro una stringa di controllo e delle variabili

NB: `disp` e **`fprintf`** non sono equivalenti, ad esempio quando vogliamo stampare dei valori complessi

Esempio

```
u = input(' inserire vettore: ');
v = input(' inserire vettore: ');
u = u(:)';
v = v(:)';
% in questo modo u e v saranno vettori riga
if (u > v)
  disp([num2str(u) ' è sempre maggiore di '
num2str(v)]);
elseif(u < v)
  disp([num2str(u) ' è sempre minore di `
num2str(v)]);
elseif(u == v)
    disp([num2str(u) ' e ' num2str(v) '
coincidono']);
else
    disp([num2str(u) ' ha valori sia maggiori che
minori di ' num2str(v)]);
end
```

Salvataggio dei Dati su File: Vantaggi

- **`save`** per file in formato .mat:
  - **`save filename`**: salva su filename.mat tutte le variabili contenute nel workspace
  - **`save('filename', 'array1' ,'array2')`**: salva su filename.mat le variabili **`array1`** e **`array2`**
- I file .mat hanno un formato compatto
- Contengono
  - Nomi, tipi e valori di ogni variabile
  - La dimensione degli array
  - e tutto ciò che serve per ripristinare il workspace
- Possono essere portati da un computer all'altro, anche con sistemi operativi diversi

Salvataggio dei Dati su File: Svantaggi

- Il formato .mat è un formato proprietario di MATLAB
- Non è utilizzabile per leggere/scrivere dati con un altro programma (e.g., editor di testo, excel)
- L'alternative è il salvataggio come file di testo con `save(fileName, varNames, ..., format)`

Es.

```
x = [1.23 3.14 6.28; -5.1 7.00 0];
save('filename.dat', 'x', '-ascii');
```

produce un file con il seguente contenuto:

```
1.2300000e+000  3.1400000e+000  6.2800000e+000
-5.1000000e+000 7.0000000e+000  0.0000000e+000
```

Caricamento Dati da File

- funzione **`load`**: carica i dati da file (formato .mat o ascii) nel workspace corrente
  - **`load filename`**: carica nello spazio di lavoro tutte le variabili nel file
  - **`load filename x y`**: carica nello spazio di lavoro solo le variabili **`x`** ed **`y`**
  - **`S = load(filename);`** carica il contenuto di filename in una struttura chiamata **`S`**
- Se filename non ha estensione viene trattato come un file .mat:
  - **`load filename.dat:`** crea una variabile di nome filename che conterrà i dati presenti in filename.dat
  - Il file deve contenere dati separati da virgole o spazi

Caricamento Dati da File Excel

- funzione **`xlsread()`**: carica i dati da un file excel (.xls o .xlsx)
  - **`xlsread(filename)`** carica tutti i dati del primo sheet di un file excel
  - **`xlsread(filename, sheet)`** carica i dati del foglio **`sheet`** contenuto in **`filename`**
  - **`xlsread(filename, sheet, range)`** carica i dati del foglio **`sheet`** all'interno del range **`range`**, specificato come una stringa con all'interno gli angoli della selezione

Es.

```
M = xlsread('dati', 'Sheet1', 'A1:D24');
```

Function Handles

Variabili funzioni: Function Handles

In Matlab esistono **variabili di "tipo funzione”**

Un valore di tipo funzione può essere assegnato a una variabile detta handle

- l’handle è una funzione: quindi all’handle possono essere passati alcuni parametri in ingresso e l’handle restituirà dei parametri in uscita
- l’handle è una variabile: quindi può essere utilizzato come **parametro attuale** di una funzione
- l’handle permette di passare una funzione (l’handle) come argomento di un’altra funzione (che in tal caso è detta «di ordine superiore»)

Assegnamento di un Valore di Tipo Funzione

Sintassi per definire l'handle ad una funzione esistente

```
f = @nome_funzione
```

- **`f`** è la variabile contentente la funzione
- **`nome_funzione`** è la funzione che vogliamo assegnare ad una variabile

Es.

```
>> seno = @sin
seno = @sin
>> seno(pi/2)
ans = 1
```

Funzioni Anonime

È possibile definire una funzione e assegnarla direttamente all'handle

- Non esiste un file che contiene la funzione
- La funzione viene detta anonima: non ha un nome proprio ma solo il nome dell'handle che la contiene

Sintassi

```
f = @(x, y, ...)<expr>
```

- **`x, y, ...`** sono i parametri della funzione
- **`<expr>`** è un'espressione che calcola il valore della funzione

Esercizio

Definire un function handle che definisce una funzione anonima per elevare al quadrato un input scalare e la si usi per calcolare il quadrato di 8

```
>> sq = @(x)(x^2)
sq = @(x)x^2
>> sq(8)
ans =  64
```

Esempi

```
% definizione della funzione inline
h = @(x)(-x)
% valutazione della funzione h nei punti 2 e 9
h(2); h(9);
% g che viene definita in maniera tale da operare su
vettori
g = @(x)(x.^2)
g(2)
g([2 : 2])
g([-2 : 2])
% è possibile "comporre" le funzioni (si da l'output
di g come input di h)
h(g([-2 : 2]))
% NB: non è possibile sommare i function handles
g + h
Undefined operator '+' for input arguments of type
'function_handle'.
```

Funzioni di Ordine Superiore

Funzioni di Ordine Superiore

- Definizione: se un parametro di una funzione `f` è un handle (cioè contiene un valore di tipo funzione) allora `f` è una **funzione di ordine superiore**
- L'handle passato come parametro consente ad `f` di invocare la funzione specificata nell'handle

Più Versatili delle Funzioni Tradizionali

```
quad = @(x)(x.^2)
% o alternativamente
function y = elevaAlQuadrato(x)
y = x.^2;
```

Vogliamo scrivere una funzione che possa usare sia `quad` sia `elevaAlQuadrato`

```
% faccio una funzione di ordine superiore
per valutare se una funzione è idempotente (se
f(f(x)) == f(x)) in un punto x (usando gli handles)
function res = controllaSeIdempotente(f , x)
if ( f(f(x)) == f(x))
    res = 1;
else
    res = 0;
end
```

Esempi di Utilizzo

```
% chiamata della funzione di ordine superiore
controllaSeIdempotente(quad, 1)

% ma non funziona se passo la funzione elevaAlQuadrato
controllaSeIdempotente(elevaAlQuadrato, 1)
Error using elevaAlQuadrato (line 2)
Not enough input arguments.

% occorre creare un handle per elevaAlQuadrato
controllaSeIdempotente(@elevaAlQuadrato, 1)
% oppure
controllaSeIdempotente(@(x)elevaAlQuadrato(x), 2)

% il nome delle variabili utilizzate per definire
function handle non conta
k = @(asd) elevaAlQuadrato(asd)
controllaSeIdempotente(k, 2)
```

Esempio: la Funzione map

Esempio: funzione **map** che applica una funzione **f** a tutti gli elementi conetenuti nel parametro **vin** e ritorna i risultati in **vout**

```
function [vout] = map(f, vin)
    for ii = 1:length(vin)
        vout(ii) = f(vin(ii));
    end
```

handle

Invoca la funzione passata come argomento

Es.

```
>> A = [1 2 3 4 5 6];
>> map(quad, A)
     ans = 1 4 9 16 25 36
```

A Cosa Serve la Funzione `map`?

- Non tutte le funzioni possono essere applicate a vettori, ad esempio:
  - la funzione per vedere se un numero è primo (codici sviluppati nella prima parte del corso)
  - la funzione per calcolare se un anno è bisestile
- Per applicare la funzione **`f`** a tutti gli elementi di un vettore:
  - scriviamo una seconda funzione o uno script che invoca la funzione **`f`**
  - facciamo un handle ad **`f`** e lo passiamo alla funzione **`map`**:

```
map(@controllaSePrimo, vett)
```

NB: La seconda opzione è decisamente preferibile perché più flessibile

Esempio: la Funzione Accumulatore

Applico la stessa funzione tra tutti gli elementi di un vettore

```
function [x] = acc(f, a, u)
    x = u;
    for ii = 1:length(a)
        x = f(x, a(ii));
    end
```

- **f** è una funzione con due argomenti, con elemento neutro **u** dell'operazione definita in **f (f(u, n) = n)**
- **f** viene applicata *cumulativamente* a tutti gli elementi di **a**:
  - applico **f** ad **a(1)** e all'elemento neutro, **f(u, a(1))**
  - applico **f** al risultato dell'operazione precedente e ad **a(2)**, ovvero **f(f(u, a(1)), a(2))**
  - fino a **f(f...f(f(u, a(1)), a(2)),..., a(end))**

A cosa Serve la Funzione Accumulatore?

Sommatoria su un vettore definita partendo dall'operazione binaria di somma

```
function [s] = sommatoria(a)
    som = @(x, y)(x+y);
    s = acc(som, a, 0);
```

Produttoria su un vettore definite partendo dall'operazione binaria di prodotto

```
function [s] = produttoria(a)
    prd = @(x, y)(x*y);
    s = acc(prd, a, 1);
```

Plot in MatLab

Funzioni Grafiche

Funzione	Scopo
`figure(fig)`	Apre una figura identificata dall'handle figNumber, se non presente definisce l'handle in maniera incrementale
`hold`	+ on/off definisce se tenere o cancellare il grafico attualmente presente nella figura alla prossima operazione di visualizzazione
`plot(x,y)`	Disegna in un riferimento cartesiano 2D le coppie di punti identificati da (x(1),y(1)),...,(x(n), y(n)), con x ed y di lunghezza n
`plot3(x,y,z)`	Disegna in un riferimento cartesiano 3D le coppie di punti identificati da (x(1),y(1),z(1)),...,(x(n) , y(n), z(n), con x, y e z di lunghezza n
`plot(x,y, frmStr)`	`frmStr` specifica il marker ed il colore usato nella visualizzazione dei punti
`imagesc(A)`	Cisualizza la matrice A come un'immagine in colormap di default. Ogni pixel viene ridimensionato per migliorare la visualizzazione
`imshow(A)`	Cisualizza un'immagine A in scale di colore (se A è di dimensione 2) o a colori nello spazio RGB (se A è di dimensione 3)
`legend(titles)`	Visualizza la legenda, usando le stringhe in titles

Diagrammi a Due Dimensioni

La funzione **`plot(x, y)`** disegna il **diagramma** cartesiano dei punti che hanno valori delle ascisse nel vettore **`x`**, delle ordinate nel vettore **`y`**

Il diagramma è la spezzata che congiunge l'insieme **dei punti**

`[x(1), y(1)], ..., [x(end), y(end)]`

rappresentanti le coordinate dei punti del piano cartesiano

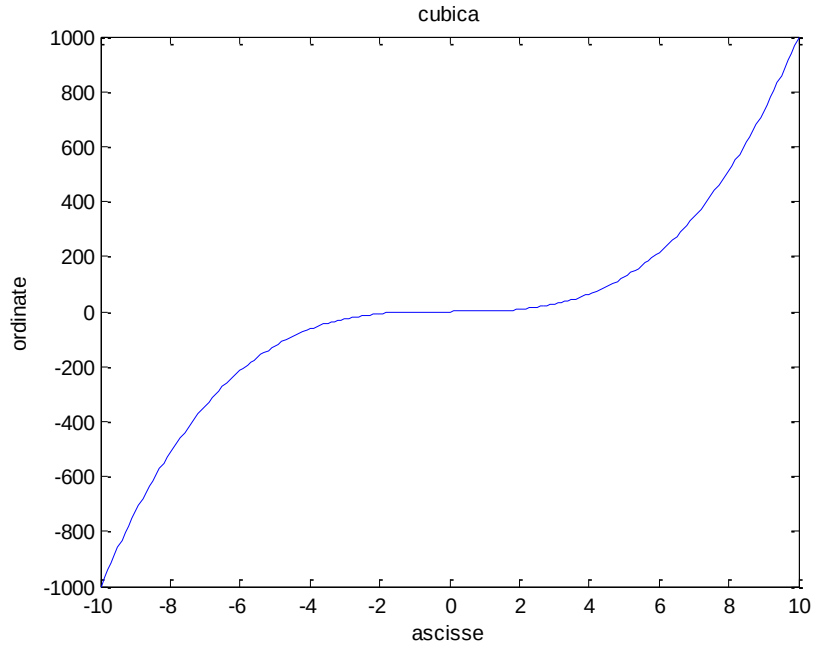
NB: `x` e `y` devono essere due vettori aventi le stesse dimensioni

È possibile specificare diversi elementi grafici (vedi **`help plot`**)

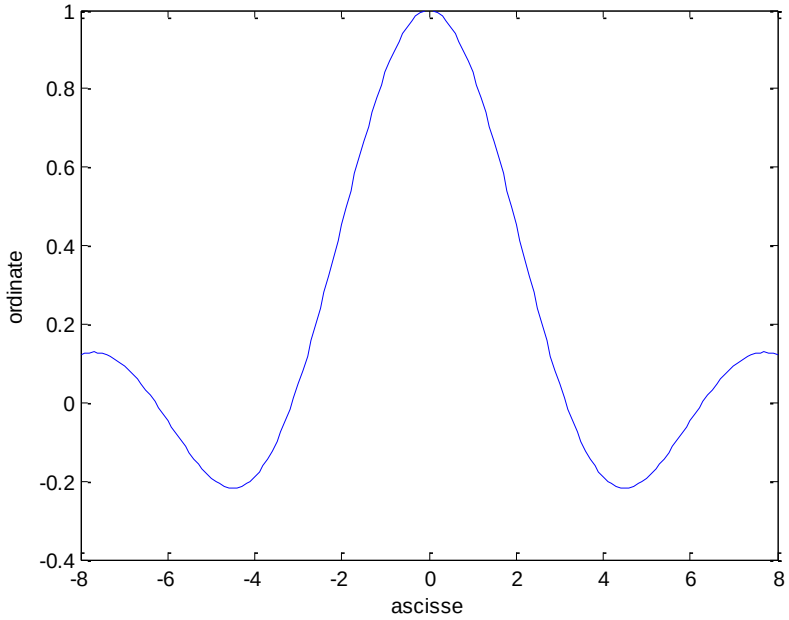
La funzione **`xlabel`** visualizza una stringa sull'asse delle ascisse, **`ylabel`** su quello delle ordinate, **`title`** per dare un titolo

Esempi: Funzioni

```
>> x = -10:0.1:10;
>> y = x.^3;
>> plot(x, y);
>> xlabel('ascisse');
>> ylabel('ordinate');
>> title('cubica');
```

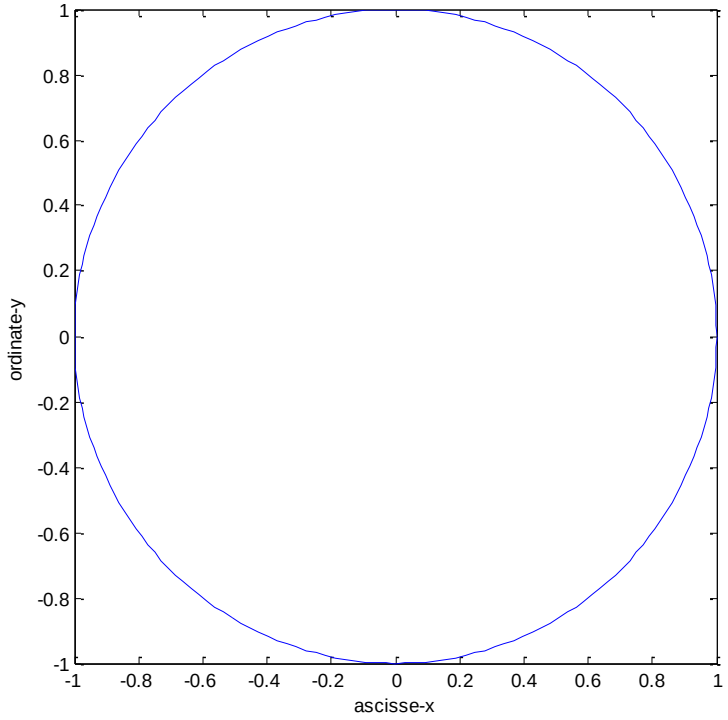



```
>> x = [-8:0.1:8];
>> y = sin (x) ./ x;
>> plot(x, y);
>> xlabel('ascisse');
>> ylabel('ordinate');
```

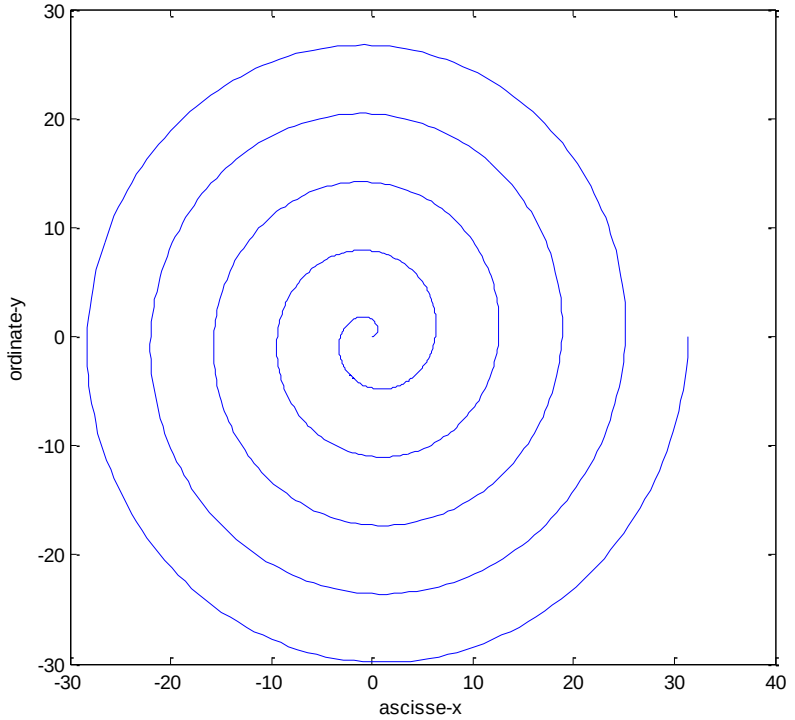


Esempi: Altri Grafici

```
>> t = [0:pi/100:2*pi];
>> x = cos(t);
>> y = sin(t);
>> plot(x, y);
>> xlabel('ascisse');
>> ylabel('ordinate');
```




```
>> t = [0:pi/100:10*pi];
>> x = t .* cos(t);
>> y = t .* sin(t);
>> plot(x, y);
>> xlabel('ascisse');
>> ylabel('ordinate');
```



Esempi

- Definire una funzione **`samplePolynomial`** che prende in ingresso:
  - un vettore di coefficienti **`polyCoeff`**
  - un vettore **`interval`** che definisce un intervallo [a, b]

e restituisce due vettori di 100 punti **`xx`** ed **`yy`** contenenti i punti che stanno sulla curva (e le cui ascisse stanno in [a,b])

$$y = C(1)x^{n-1} + C(2)x^{n-2} + \ldots + C(n-1)x^1 + C(n)$$

- Utilizzare **`samplePolynomial`** per calcolare i punti delle seguenti curve (in un intervallo [-10 10]) e visualizzarlo:

$$y = x - 1;$$
$$y = 2x^2 + x - 12;$$
$$y = -0.1x^3 + 2x^2 - 10x - 12$$

- Evidenziare, per ogni valore di x, la curva avente y maggiore

Soluzione (Funzione)

```
function [xx, yy] =
samplePolynomial(polyCoeff, interval)

% per essere certi che a <= b
a = min(interval);
b = max(interval);

xx = [a  : (b-a) / 99 : b];
% oppure xx = linspace(a , b, 100)
yy  = zeros(size(xx));

for ii = 1:1:length(polyCoeff)
    yy = yy + polyCoeff(ii) * ...
     xx.^(length(polyCoeff) - ii);
end
```

Soluzione (Calcolo Curve)

```
% definisco intervallo e coefficienti
interval = [-10, 10];
rettaCoeffs = [1, -1];
parabolaCoeffs = [2, 1, -12] ;
cubicaCoeffs = [-0.1, 2, -10, -12];

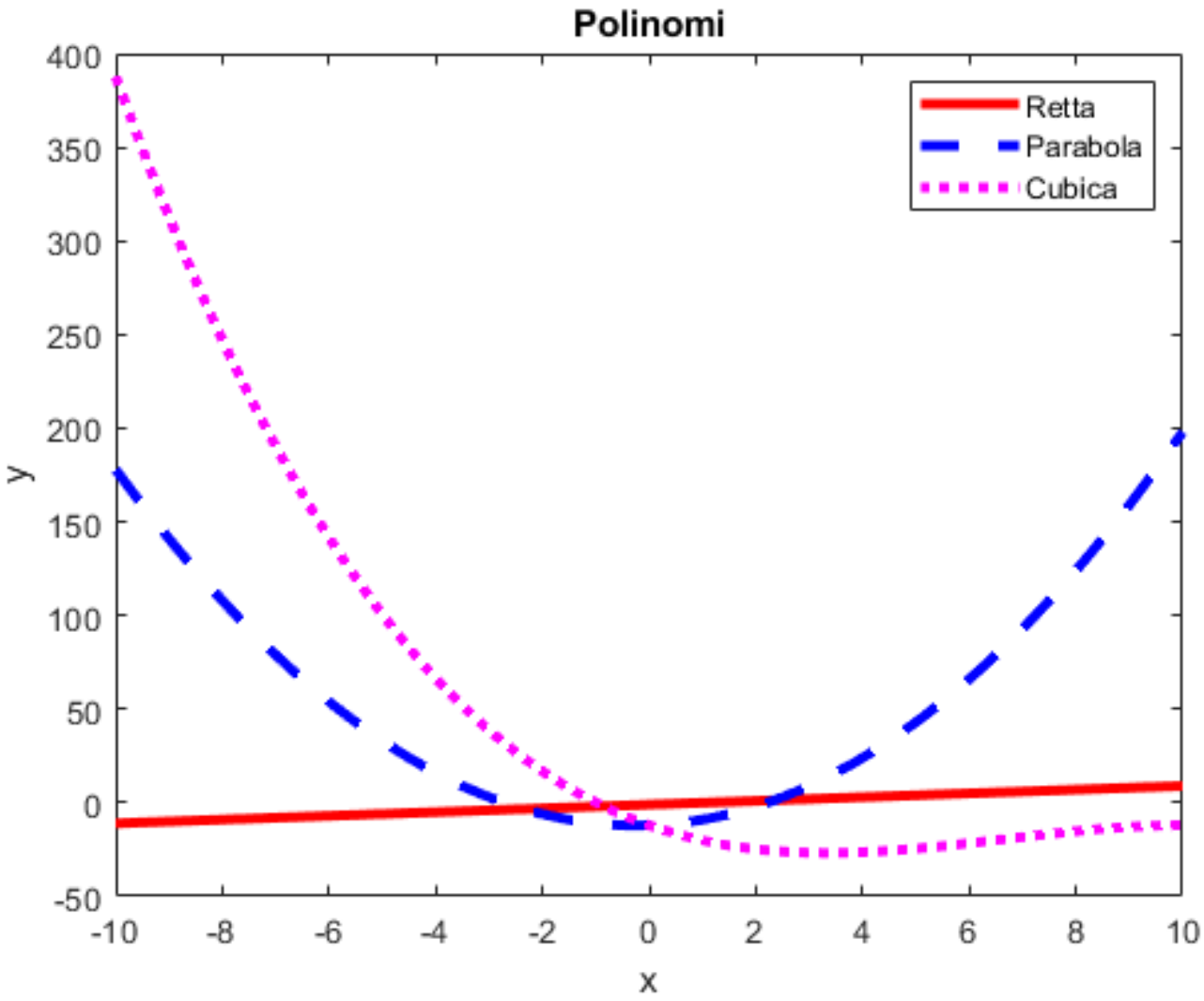
% calcola i valori dei polinomi
[rx, ry] = samplePolynomial(rettaCoeffs,
interval);
[px, py] = samplePolynomial(parabolaCoeffs,
interval);
[cx, cy] = samplePolynomial(cubicaCoeffs,
interval);
```

Soluzione (Plot)

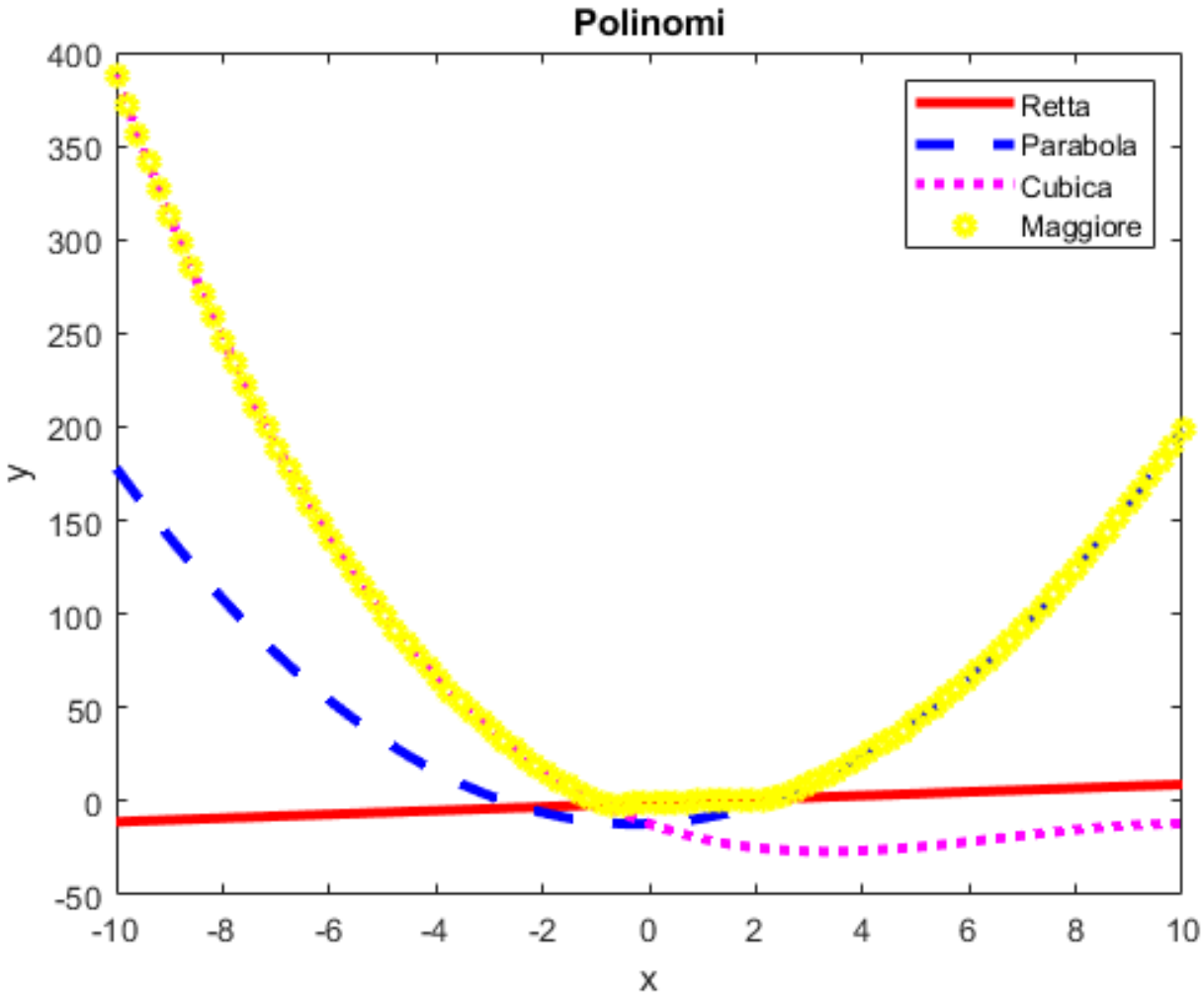
```
figure();
plot(rx, ry, 'r-', 'LineWidth', 3);
hold on;
plot(px, py, 'b--', 'LineWidth', 3);
plot(cx, cy, 'm:', 'LineWidth', 3);
hold off;

legend('retta', 'parabola', 'cubica');
xlabel('x');
ylabel('y');
title('Polinomi');
```

Risultato


Polinomi
Retta
Parabola
Cubica
y
x
400
350
300
250
200
150
100
50
0
-50
-10
-8
-6
-4
-2
0
2
4
6
8
10

Come Ottengo Questo?



Disegno i Punti della Curva più Alta

```
% determina i punti ad ordinata maggiore
indx_r = find(ry > py & ry > cy);
indx_p = find(py > ry & py > cy);
indx_c = find(cy > py & cy > ry);
hold on
plot(rx(indx_r), ry(indx_r), 'yo',
     'LineWidth', 3, 'MarkerSize', 5);
plot(px(indx_p), py(indx_p), 'yo',
     'LineWidth', 3, 'MarkerSize', 5);
plot(cx(indx_c), cy(indx_c), 'yo',
     'LineWidth', 3, 'MarkerSize', 5);
legend('Retta', 'Parabola', 'Cubica',
     'Maggiore');
```

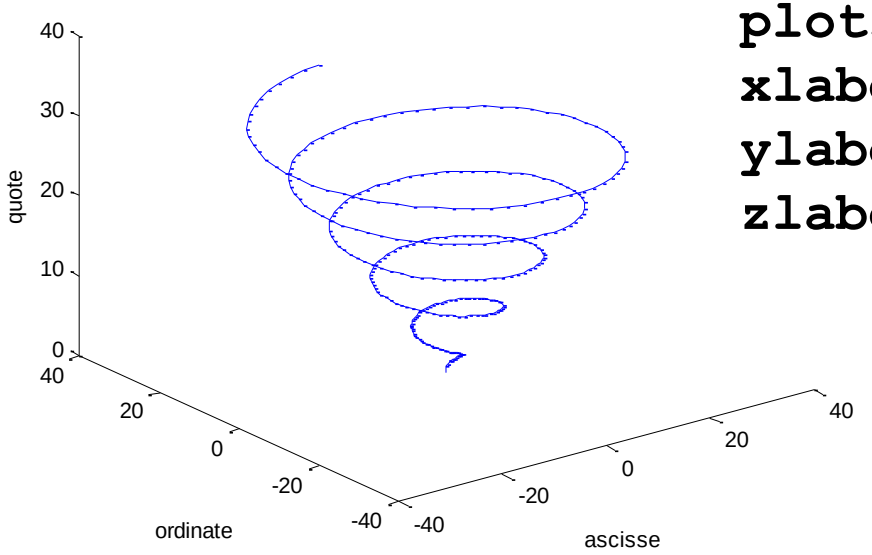
Plot in Tre Dimensioni

Generalizzazione del diagramma a due dimensione: insieme di terne di coordinate

`plot3(x, y, z)` disegna un diagramma cartesiano con **`x`** come ascisse, **`y`** come ordinate e **`z`** come quote

Posso ancora usare funzioni come **`xlabel, ylabel, zlabel, title`**

Es.




```
t = 0:0.1:10*pi;
plot3 (t.*sin(t), t.*cos(t), t);
xlabel('ascisse');
ylabel('ordinate');
zlabel('quote');
```

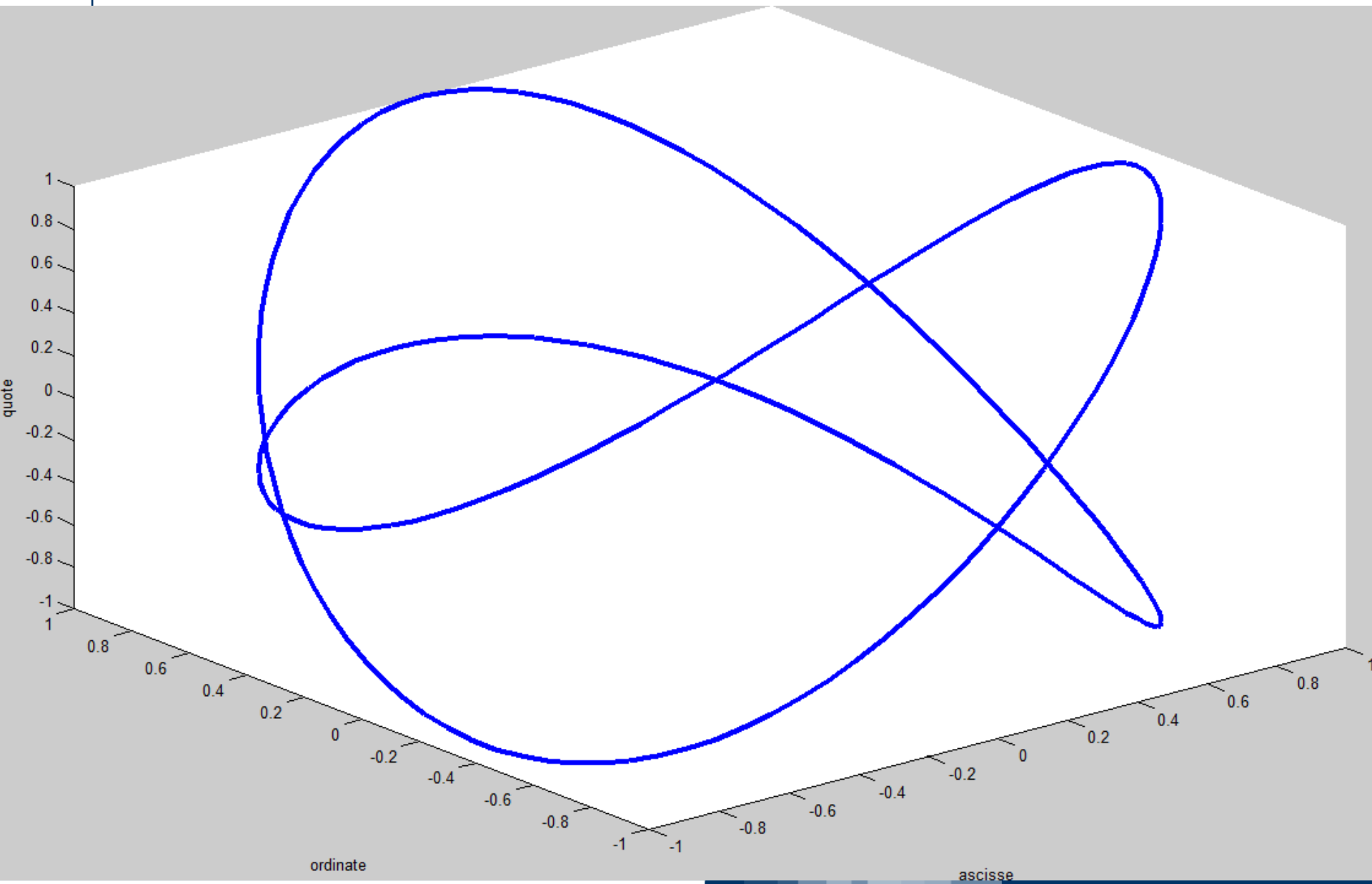
La Funzione `linspace`

- **`linspace(a, b, n)`:** crea un vettore di **`n`** punti equispaziati tra **`a`** e **`b`**
- **`plot`** restituisce un handle, una variabile di riferimento per poter accedere nuovamente all'insieme di punti disegnato
- **`set(plot_handle, 'Property Name', PropertyVal)`** permette di modificare le proprietà del plot

Es.

```
t = linspace(0, 4*pi ,200);
plot_hnd = plot3(sin(t), cos(t), cos(3/2*t));
set(plot_hnd, 'LineWidth', 3);
xlabel('Ascisse');
ylabel('Ordinate');
zlabel('Quote');
```

Risultato

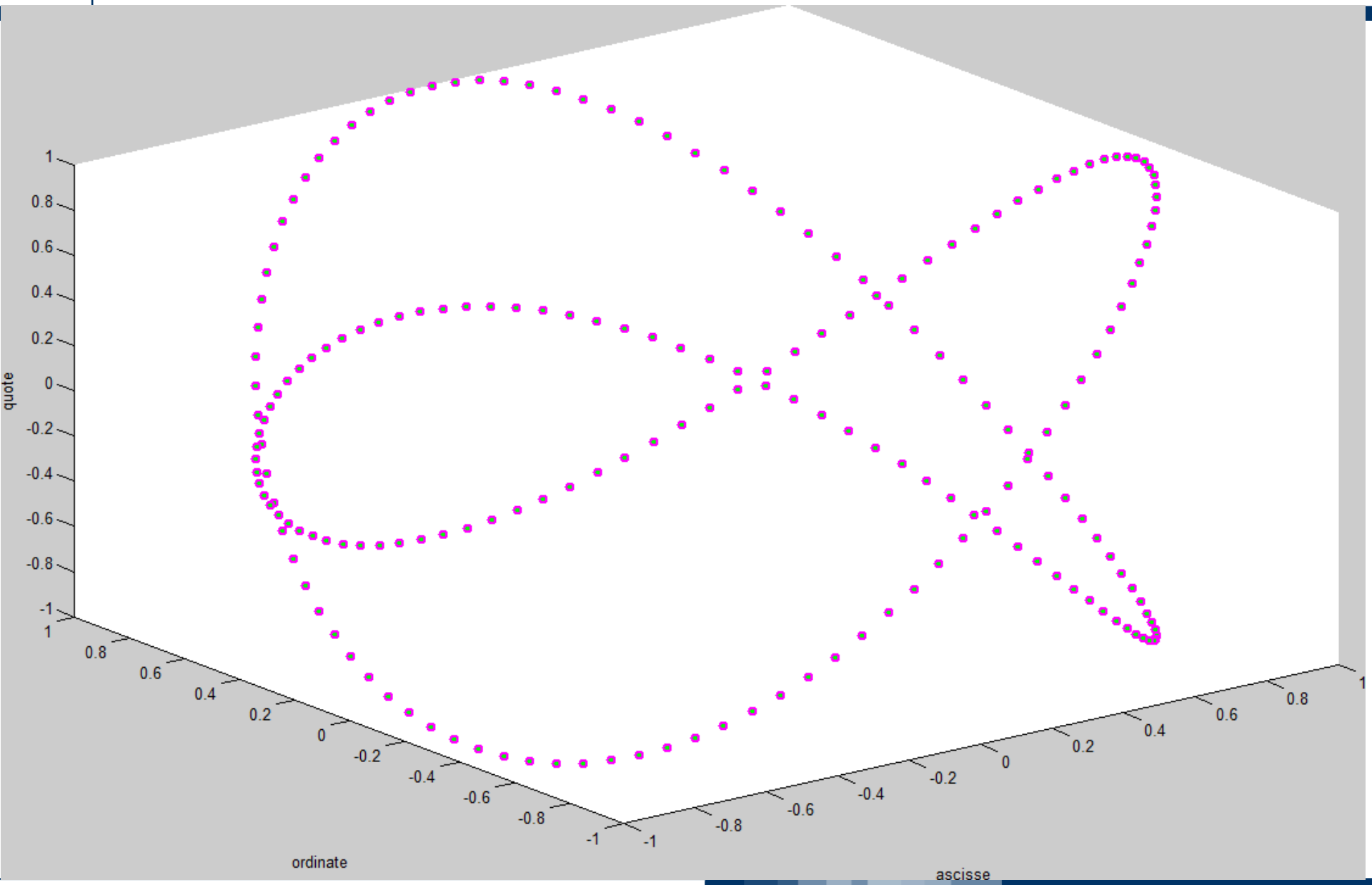

quote
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
ordinate
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
ascisse
-1
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8

Cosa fa Questo Codice?

```
t = linspace(0, 4*pi ,200);
plot_hnd = plot3(sin(t),cos(t),cos(3/2 *t));
set(plot_hnd, 'LineWidth', 3);
xlabel('Ascisse');
ylabel('Ordinate');
zlabel('Quote');

set(plot_hnd ,'LineStyle', 'none', 'Marker',
'o', 'MarkerFaceColor', [0 1 0], ...
'MarkerEdgeColor', [1 0 1], ...
'MarkerSize', 5, 'LineWidth', 1.5);
```

Risultato


quote
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
ordinate
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
ascisse
-1
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8
1

Superfici

Come si disegna una superficie che rappresenta una funzione a due variabili $z = f(x,y)$?

Occorre definire il dominio che non è più un intervallo in una retta ma una porzione del piano

La funzione **`mesh(xx, yy, zz)`** genera superficie, a partire da tre argomenti:

- **`xx`** contiene le ascisse
- **`yy`** contiene le ordinate
- **`zz`** contiene le quote

`xx` e **`yy`** sono matrici che identificano una griglia in corrispondenza del quale per **`zz`** rappresenta il valore della funzione

Funzione `meshgrid`

Le due matrici, **`xx`** e **`yy`**, si possono costruire mediante la funzione **`meshgrid(x,y)`**

```
[xx, yy] = meshgrid(x, y)
```

- **`x`** e **`y`** sono due vettori
- **`xx`** e **`yy`** sono due matrici entrambe di length(y) righe e length(x) colonne
- la prima, **`xx`**, contiene, ripetuti in ogni riga, i valori di **`x`**
- la seconda, **`yy`** , contiene, ripetuti in ogni colonna, i valori di **`y`** trasposto

Risultato di `meshgrid`

```
[xx, yy] = meshgrid([-3 : 3], [-4 : 4]);
```

xx =

-3	-2	-1	0	1	2	3
-3	-2	-1	0	1	2	3
-3	-2	-1	0	1	2	3
-3	-2	-1	0	1	2	3
-3	-2	-1	0	1	2	3
-3	-2	-1	0	1	2	3
-3	-2	-1	0	1	2	3
-3	-2	-1	0	1	2	3
-3	-2	-1	0	1	2	3

yy =

-4	-4	-4	-4	-4	-4	-4
-3	-3	-3	-3	-3	-3	-3
-2	-2	-2	-2	-2	-2	-2
-1	-1	-1	-1	-1	-1	-1
0	0	0	0	0	0	0
1	1	1	1	1	1	1
2	2	2	2	2	2	2
3	3	3	3	3	3	3
4	4	4	4	4	4	4

È possibile quindi valutare una funzione di queste due matrici, e.g., zz = xx + yy , e disegnarla mediante mesh

Esempio

```
x = [1, 3, 5];
y = [2, 4];
[xx, yy] = meshgrid(x, y);
zz = xx + yy;
mesh(xx, yy, zz);
xlabel('Ascisse (x)');
ylabel('Ordinate (y)');
```

```
>> xx
xx =
   1   3   5
   1   3   5
```

```
>> yy
yy =
   2   2   2
   4   4   4
```

Punti (x,y)

(1,2) (3,2) (5,2)

(1,4) (3,4) (5,4)

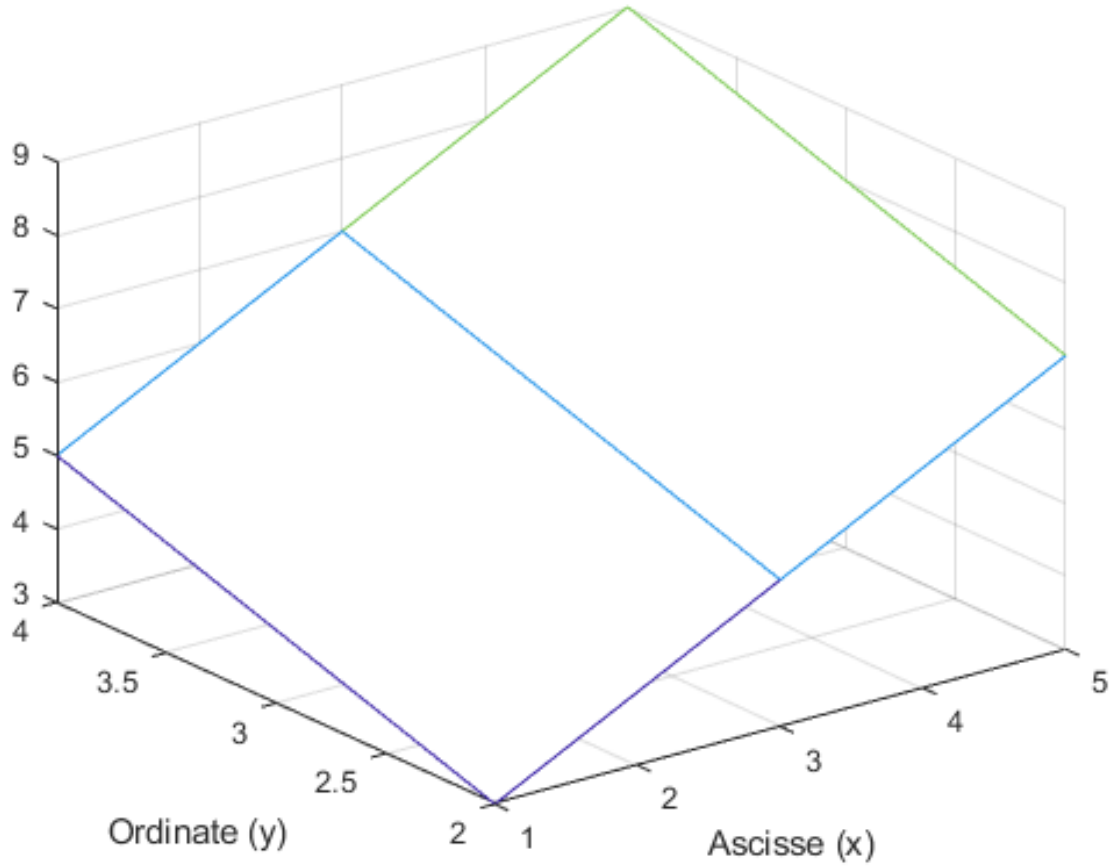
Coordinate (x,y,z)

(1,2,3) (3,2,5) (5,2,7)

(1,4,5) (3,4,7) (5,4,9)

```
>> zz
zz =
   3   5   7
   5   7   9
```

Risultato



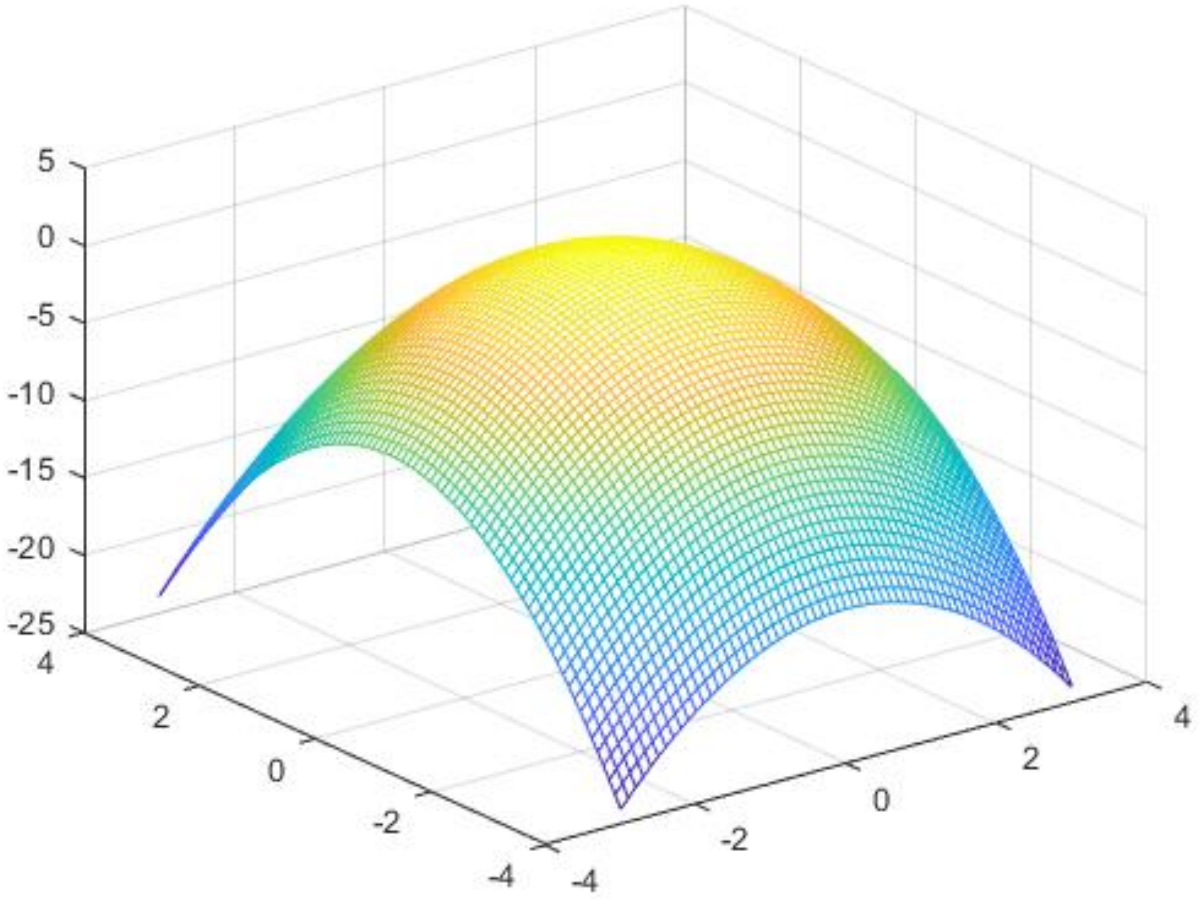
mesh e Function Handles

```
[xx, yy] = meshgrid([-3:0.1:3], [-4:0.1:4]);
f = @(x, y)(1 - x.^2 - y.^2);

figure();
aa = mesh(xx, yy, f(xx, yy));
```

- Mesh unisce i punti con delle linee colorate
- Di default il colore indica il valore della quota

Risultato



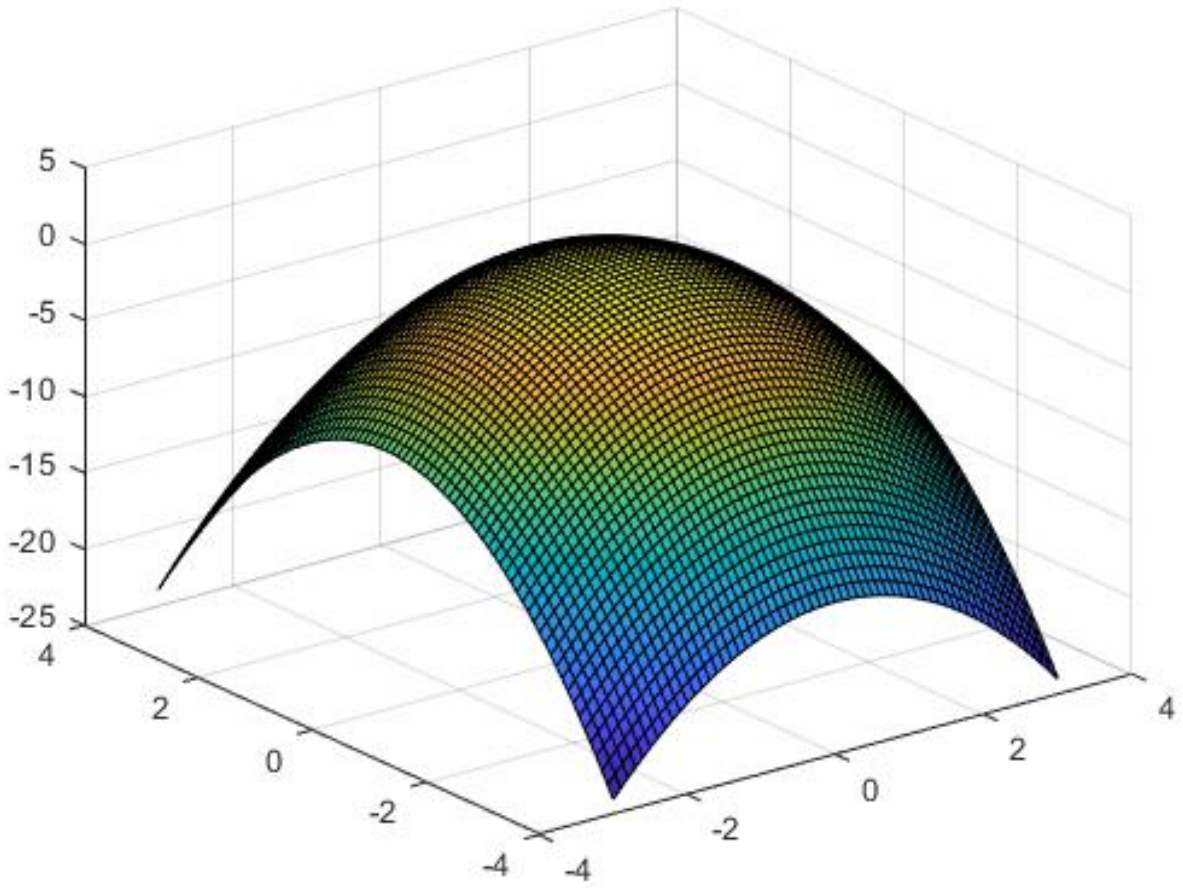
Superfici Piene: `surf`

```
[xx, yy] = meshgrid([-3:0.1:3], [-4:0.1:4]);
f = @(x, y)(1 - x.^2 - y.^2);

figure();
aa = surf(xx, yy, f(xx, yy));
```

- Surf riempie le regioni tra le linee con del colore

Risultato



hold on

- Le superfici vengono visualizzate su un grafico 3D
- È quindi possibile aggiungere degli elementi in sovraimpressione utilizzando le funzioni:
  - **`plot3()`**, **`mesh()`**, altre funzioni grafiche quali **`surf()`**, etc.
- Per sovrascrivere ad un grafico usare la funzione **`hold on`** e **`hold off`** quando si ha terminato

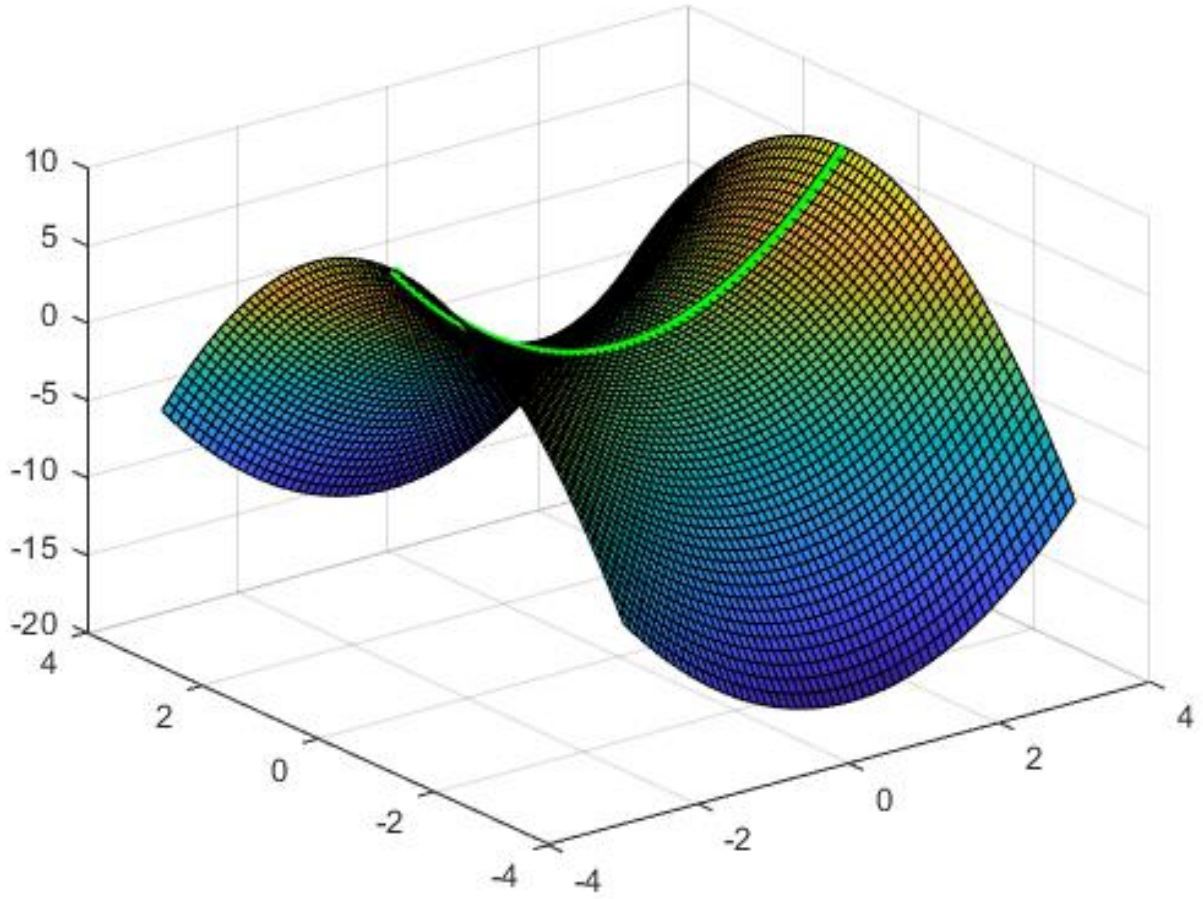
Esercizio

Es. disegnare in sovraimpressione alla quadrica

$$z = x^2 - y^2 \text{ la curva } \begin{cases} z = x^2 \\ y = 0 \end{cases}$$

```
[xx, yy] = meshgrid([-3:0.1:3], [-4:0.1:4]);
f = @(x, y)(x.^2 - y.^2);
figure();
aa = surf(xx, yy, f(xx, yy));
hold on;
x = xx(1, :);
y = zeros(size(x));
bb = plot3(x, y, f(x,y), 'g-');
set(bb, 'LineWidth', 3);
hold off;
```

Risultato



Ultimo Grafico

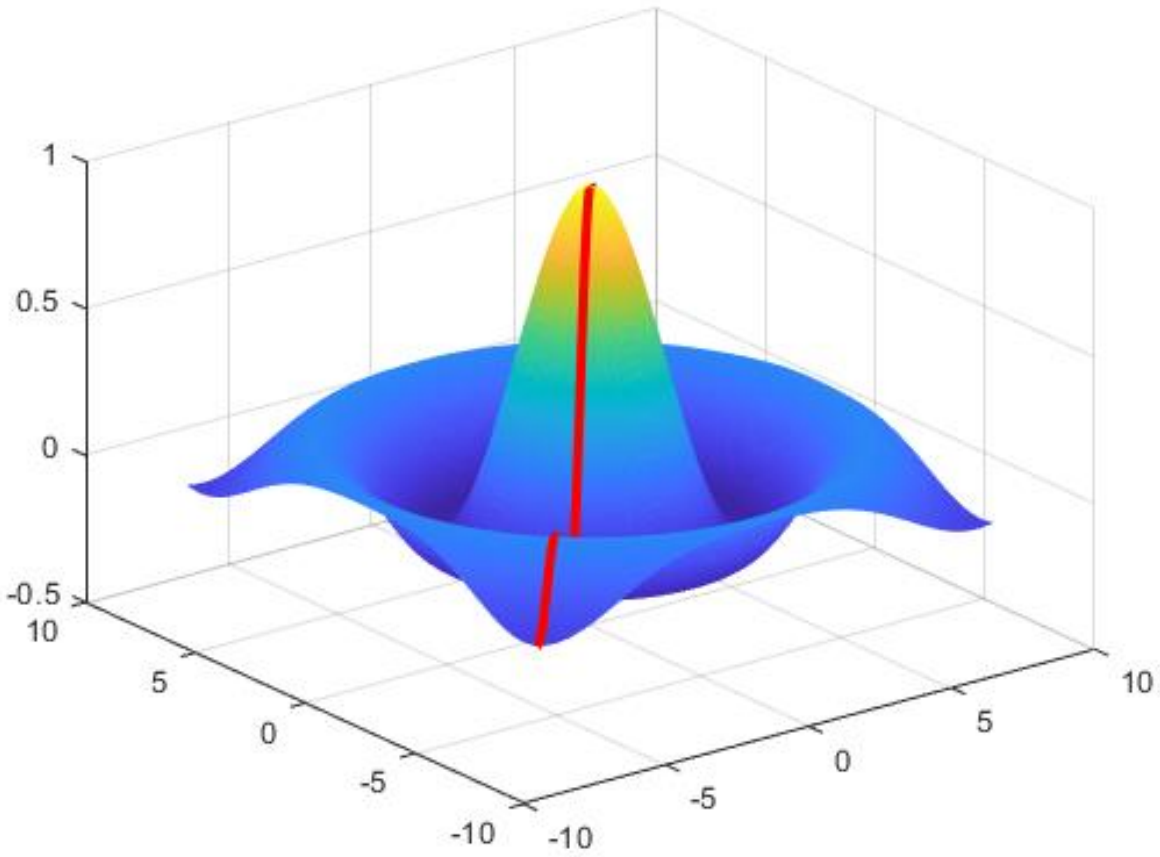
Disegnare la funzione «mexican hat»:

$$z = \frac{\sin\left(\sqrt{x^2 + y^2}\right)}{\sqrt{x^2 + y^2}}$$

e una curva su questa funzione passante per l'origine

```
tx = [-8:0.1:8]; ty = tx;
[xx, yy] = meshgrid(tx, ty);
f = @(x,y)(sin(sqrt(x.^2 + y.^2))./sqrt(x.^2
+ y.^2));
figure();
aa = surf(xx, yy, f(xx, yy));
hold on
plot3(tx, tx, f(tx,tx),'r-','LineWidth', 3);
set(aa,'edgecolor','none')
```

Superfici: Esempi (3)



Strutture in Matlab

Structure Array (Array di Strutture)

- Una struttura è un tipo di dato composto da elementi individuali possibilmente non omogenei
- Ogni elemento individuale è chiamato *campo* ed ha un nome
- Una struttura può avere campi di tipo diverso
- E' possibile (naturale) creare array di strutture
- **Creazione di una struttura** (e di array di strutture):
  - Utilizzando la funzione struct()
  - Assegnamento diretto dei valori ai campi (e contestuale definizione dei campi)

Creazione di una Struttura

- Creazione di una struttura :

```
var = struct(nomecampo1, valorecampo1, ...)
```

  - **var** nome della struttura
  - **nomecampo1** campo della struttura riempito con **valorecampo1**
  - **...** altri campi

Es.

```
studente = struct('nome', 'Giovanni', 'eta', 24);
```

Creazione Tramite Assegnamento dei Valori

- Non definisco in una volta sola i campi della struttura
- Riempio i campi ad uno ad uno
- Posso aggiungere campi ad una struttura quando voglio

Es.

```
studente.nome = 'Giovanni';
studente.eta = 24;
```

- Anche in MatLab posso utilizzare l'operatore **dot** per accedere a i campi di una struttura, sia per leggere che per modificare

Array di Strutture

- Tutte le strutture dell'array devono avere gli stessi campi ovvero **l'array deve essere omogeneo (compost dale stesse strutture)**
- É possibile far diventare **`studente`** un array di strutture, accodando un altro elemento in **`studente(2)`**

Es.

```
studente(2).nome = 'Giulia';
studente(2).eta = 22;
```

- É possibile assegnare solo alcuni campi a **`studente(2)`**: i campi non assegnati rimangono vuoti

Aggiunta di Campi ad Array di Strutture

- Il campo esami viene aggiunto a tutte le strutture che fanno parte del vettore
- Avrà un valore iniziale per la struttura a cui viene assegnato, e sarà vuoto per tutti gli altri elementi dell'array

Es.

```
studente(2).esami = [20 25 30];
```

Solo **`student(2)`** avrà il campo **`esami`** pieno, le altre strutture invece avranno quel campo vuoto

Array di Strutture Innestati

- Un campo di una struttura può essere anch'esso una strutture
- È quindi possibile avere un campo che è, di nuovo, una struttura o un array di strutture

Es.

```
studente(1).corso(1).nome = 'InformaticaB';
studente(1).corso(1).docente = 'Von Neumann';
studente(1).corso(2).nome = 'Matematica';
studente(1).corso(2).docente = 'Eulero';
```

Esercizio

Si sviluppi uno script matlab che acquisisce da tastiera i dati relativi ad un numero arbitrario di rilievi altimetrici e che quindi stampa a video l'altitudine media di tutti i rilievi che si trovano nell'intervallo

- latitudine [30, 60]
- longitudine [10, 100]

Soluzione

```
n = input(['quanti rilievi? ']);
for ii = 1 : n
    s(ii).altezza = input(['altezza rilievo nr ',
num2str(ii), ' ']);
    s(ii).latitudine= input(['latitudine rilievo nr
', num2str(ii), ' ']);
    s(ii).longitudine= input(['longitudine rilievo nr
', num2str(ii), ' ']);
end
% creo dei vettori con i valori dei campi
LAT = [s.latitudine]; LON = [s.longitudine];
ALT = [s.altezza];
% definizione del sottovettore da estrarre da altezza
latOK = (LAT > 30) & (LAT <60);
lonOK = (LON> 10) & (LON<100);
posOK =latOK & lonOK;
mean(ALT(posOK))
```

Modificare i Campi di un'Array di Strutture

- È possibile estrarre tutti gli elementi che appartengono allo stesso campo di un array di strutture

Es. **`[rilievi.altezza]`**

- Tuttavia non è possibile modificare con una sintassi simile i valori sui campi di tutti gli elementi dell'array.

Es. **NON** è possibile fare

`[rilievi.altezza] = [rilievi.altezza] + 10`

per sommare 10 a tutte le altezze di rilievi perché la parte a sx dell'uguale non è una variabile

- Occorre quindi procedere elemento per elemento

Modificare i Campi di un'Array di Strutture

Es. somma di uno scalare a tutte le altezze dei rilievi

```
function ril = sommaAlt(ril, offset)
for ii = 1 : length(ril)
    ril(ii).alt = ril(ii).alt + offset;
end
```

- Tuttavia la funzione **`sommaAlt`** non può modificare «da sola» una variabile **`rilievi`** nel workspace
- È necessario a sovrascrivere **`rilievi`** il parametro restituito da **`sommaAlt`**

Es. **`rilievi = sommaAlt(rilievi, 10)`**