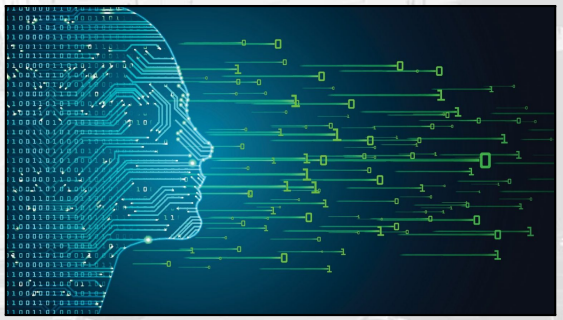
Programmazione Informatica
9. MATLAB, parte 4: altre strutture dati e grafici

Riccardo Zamolo
rzamolo@units.it

Università degli Studi Trieste
Ingegneria Civile e Ambientale



A.A. 2020-21

- Precedentemente abbiamo usato matrici e vettori come strutture dati per operare su una sequenza ordinata di dati omogenei, cioè dello stesso tipo: numeri (`double`, `single`,...) o caratteri (`char` e `string`).
- È possibile in MATLAB operare su una sequenza ordinata di dati non omogenei mediante *cell arrays*, ossia array di celle, ognuna delle quale può contenere qualsiasi tipo di dato.
- Inizializzazione esplicita:

  `c = cell( m1 , ... , mD )`

  per allocare un array di celle di dimensione `m1`$\times\cdots\times$`mD`, dove ogni cella è inizializzata di default con una matrice vuota di tipo double. Ad esempio, un vettore (array monodimensionale) di `m` celle si inizializza con `cell(m)`, una matrice (array bidimensionale) di `m`$\times$`n` celle si inizializza con `cell(m,n)`:

```
>> c = cell( 2 , 3 )
c =
  2×3 cell array
   {0×0 double}    {0×0 double}    {0×0 double}
   {0×0 double}    {0×0 double}    {0×0 double}
```

- Le celle di un cell array sono di un nuovo tipo, `cell`:

```
>> class( c(1) )
ans =
    'cell'
```

- Si può anche definire direttamente un cell array a partire dal contenuto che si vuole per ogni cella, usando `','` e `';'` per separare le celle per riga e per colonna, esattamente come per la definizione di una matrice/vettore, ma usando le parentesi graffe:

```
>> Bevande = { 'Acqua' , 1.5 ; ...
               'Thè'   , 1.5 ; ...
               'Birra' ,  30 }
```

- L'accesso e l'assegnazione avviene attraverso gli indici, esattamente come per qualsiasi altro array, tenendo però conto che gli elementi di un cell array sono di tipo `cell`, mentre è il contenuto di ogni cella ad essere di tipo arbitrario:

Accesso con parentesi tonde:

```
>> Bevande(2,1)
ans =
  1×1 cell array
    {'Thè'}
```

Accesso con parentesi graffe:

```
>> Bevande{2,1}
ans =
    'Thè'
```

- Mediante cell array è possibile operare su sequenze ordinate di function handle, cosa non possibile mediante semplici array. Per esempio, per definire la base $P_2 = \{1, x, x^2\}$ dello spazio vettoriale dei polinomi di secondo grado utilizzeremo il seguente cell array di function handle:

```
>> P_2 = { @(x) 1 , @(x) x , @(x) x .^ 2 }
P_2 =
  1×3 cell array
    {@(x)1}    {@(x)x}    {@(x)x.^2}
```

- *Funzioni linearmente indipendenti.* n funzioni $f_1(x), \ldots, f_n(x)$ definite sul dominio $\mathcal{D}$ sono linearmente indipendenti se la funzione nulla $f(x) = 0$ si può ottenere solo come combinazione lineare con coefficienti a_i nulli:

$$a_1 f_1(x) + \ldots + a_n f_n(x) = 0 \quad \forall x \in \mathcal{D} \quad \Leftrightarrow \quad a_i = 0$$

- Per verificare se n funzioni $f_1(x), \ldots, f_n(x)$ sono linearmente indipendenti basta perciò verificare che non esista nessuna combinazione di coefficienti $a_i \neq 0$ che genera la funzione nulla $f(x) = 0$. Scegliamo quindi $m > n$ punti $x_1, \ldots, x_m \in \mathcal{D}$ e scriviamo il sistema:

$$\begin{cases} a_1 f_1(x_1) + \cdots + a_n f_n(x_1) = 0 \\ a_1 f_1(x_2) + \cdots + a_n f_n(x_2) = 0 \\ \quad \vdots \\ a_1 f_1(x_m) + \cdots + a_n f_n(x_m) = 0 \end{cases}$$

che scriveremo $\mathbf{Fa} = \mathbf{0}$, dove $\mathbf{a} = (a_i)$ è il vettore colonna degli n coefficienti ed $\mathbf{F}$ è la matrice $m \times n$

$$\mathbf{F} = \begin{bmatrix} f_1(x_1) & \cdots & f_n(x_1) \\ \vdots & \ddots & \vdots \\ f_1(x_m) & \cdots & f_n(x_m) \end{bmatrix}$$

- Se le n funzioni $f_1(x), \ldots, f_n(x)$ sono linearmente dipendenti, esisterà una soluzione non nulla al precedente sistema lineare (sovradeterminato perchè $m > n$), cioè $\ker(\mathbf{F})$ avrà dimensione maggiore o uguale a 1, ossia non sarà vuoto. Quindi basterà calcolare $\ker(\mathbf{F})$ per un numero sufficiente di punti $x_1, \ldots, x_m \in \mathcal{D}$, con $m > n$.
- Scrivere una funzione che prenda in input una base di n funzioni, definite mediante un cell array lungo n, ed un intervallo finito $[a, b] \subseteq \mathcal{D}$ del dominio $\mathcal{D}$, e determini se sono linearmente indipendenti:

f_indipendenti.m

```
% Input: f, cell array dei function handle delle funzioni f_i
%        ab, intervallo dentro il dominio
% Output: I, booleano che indica se le f_i sono linearmente indipendenti
function I = f_indipendenti( f , ab )
  n = length( f ) ; % numero di funzioni
  m = 2*n ;         % numero di punti
  x = linspace( ab(1) , ab(2) , m )' ;
  F = zeros( m , n ) ; % allocazione matrice F
  for i = 1 : n
    F(:,i) = f{i}(x) ; % valutazione f_i( x1 , ... , xm )
  end
  kF = null( F ) ; % ker( F )
  I  = size( kF , 2 ) == 0 ;
end
```

- Utilizzo con l'insieme di funzioni $P_2 = \{1, x, x^2\}$:

```
% Definizione del cell array delle funzioni e verifica dell'indipendenza
P_2   = { @(x) 1+0*x , @(x) x , @(x) x .^ 2 } ;
I_P_2 = f_indipendenti( P_2 , [0 1] ) ;

% Output formattato
output_f_indipendenti( P_2 , I_P_2 ) ;
```

dove la funzione **output_f_indipendenti** serve ad ottenere l'output formattato della verifica dell'indipendenza:

output_f_indipendenti.m

```
% Input: f, cell array di function handle delle funzioni f_i
%        I, booleano che indica se le f_i sono indipendenti
% Output: nessuno
function output_f_indipendenti( f , I )
  fprintf( 'Le funzioni\n' ) ;
  disp(f) ;
  verbo = { 'non sono' , 'sono' } ;
  fprintf( [ verbo{I+1} ' linearmente indipendenti.\n' ] ) ;
end
```

ottenendo:

```
Le funzioni
    {@(x)1+0*x}    {@(x)x}    {@(x)x.^2}

sono linearmente indipendenti.
```

- Utilizzo con l'insieme di funzioni $H = \{\sin x, \cos x, \sin(x + \pi/4)\}$:

```
% Definizione del cell array delle funzioni e verifica dell'indipendenza
H = { @(x) sin(x) , @(x) cos(x) , @(x) sin(x+pi/4) } ;
I_H = f_indipendenti( H , [0 2*pi] ) ;

% Output formattato
output_f_indipendenti( H , I_H ) ;
```

ottenendo:

```
Le funzioni
    {@(x)sin(x)}    {@(x)cos(x)}    {@(x)sin(x+pi/4)}

non sono linearmente indipendenti.
```

che è corretto in quanto $\sin(x + \pi/4) = \frac{1}{\sqrt{2}} \sin x + \frac{1}{\sqrt{2}} \cos x$.

- Utilizzo con l'insieme $H_2 = \{1, \cos x, \cos(2x), (\cos x + 1)^2\}$:

```
H_2 = { @(x) 1+0*x , @(x) cos(x) , @(x) cos(2*x) , @(x) (cos(x)+1) .^ 2 } ;
I_H_2 = f_indipendenti( H_2 , [0 2*pi] ) ;
```

ottenendo che le funzioni non sono linearmente indipendenti in quanto $(\cos x + 1)^2 = \frac{3}{2} + 2\cos x + \frac{1}{2}\cos(2x)$.

- Abbiamo visto che nel caso di cell array è possibile operare su sequenze ordinate di dati di tipo arbitrario, utilizzando quindi uno o più indici per accedere o assegnare una (o più) celle.
- In certi casi risulta più comodo accedere ad una sequenza di dati non attraverso un indice ma attraverso un nome identificativo per ciascun elemento. Abbiamo già visto questa struttura dati, detta record, dove ogni elemento è detto *campo* ed è identificato da un nome e non da un indice. In MATLAB questo tipo di dato è chiamato **`struct`**.
- Uno **`struct`** può essere definito direttamente a partire dai suoi campi e dal nome di ogni campo:

  ```
  s = struct(nome_campo1,valore1,...,nome_campoN,valoreN)
  ```

  dove i **`nome_campo`** sono testuali (vettori di caratteri o stringhe) e i **`valore`** possono essere di tipo arbitrario, come per le celle nei cell array.

```
>> bevanda = struct( 'tipologia' ,      'amaro' , ...
                     'marca' , 'Montenegro' , ...
                  'contenuto' ,       0.750  , ...
                 'gradazione' ,          23  )
bevanda =
  struct with fields:
     tipologia: 'amaro'
         marca: 'Montenegro'
     contenuto: 0.7500
    gradazione: 23
```

- L'accesso o l'assegnazione di un valore di un campo di uno `struct` avviene mediante la notazione con il punto, `nome_struct.nome_campo`:

```
q = bevanda.contenuto ;   % accesso
bevanda.contenuto = 0.5 ; % assegnazione
```

- Grazie all'uso dei nomi identificativi di ogni campo, l'uso degli `struct` risulta molto comodo per manipolare in maniera molto chiara un determinato insieme di dati utilizzando un'unica variabile contenitore.
- Questo impacchettamento di dati eterogenei sotto un'unica entità di tipo `struct` semplifica e rende molto più chiara anche la chiamata di funzioni.
- Ad esempio, consideriamo due `struct`, `b1` e `b2`, che descrivono due bevande utilizzando per entrambi gli stessi 4 campi considerati precedentemente, e supponiamo di voler scrivere una funzione che prenda in ingresso questi due `struct` e restituisca un altro `struct`, della stessa tipologia, che descriva il mix tra le due bevande iniziali:

`mix_bevande.m`

```
% Input: b1 e b2, struct bevande
% Output: b_mix, struct bevanda mix di b1 e b2
function b_mix = mix_bevande( b1 , b2 )
  b_mix = b1 ;
  b_mix.tipologia  = unisci_nomi( b1.tipologia , b2.tipologia ) ;
  b_mix.marca      = unisci_nomi( b1.marca     , b2.marca     ) ;
  b_mix.contenuto  = b1.contenuto + b2.contenuto ;
  b_mix.gradazione = ( b1.contenuto * b1.gradazione + ...
                       b2.contenuto * b2.gradazione ) / b_mix.contenuto ;
end
```

- La funzione **mix_bevande** ha solo 2 variabili di tipo **struct** in ingresso ed una sola in uscita; le operazioni svolte dalla funzione sono molto chiare grazie all'uso dei nomi identificativi per tutti i campi degli **struct** considerati.
- Se non si fossero usati gli **struct**, avremmo avuto 8 variabili in input e 4 variabili in output, con minore chiarezza e maggiore probabilità di errore. Eventuali aggiunte successive di altri campi agli **struct** impiegati non richiedono alcuna modifica nella chiamata della funzione.
- Utilizzo:

```
M = struct( 'tipologia' ,       'amaro' , ...
            'marca' , 'Montenegro' , ...
          'contenuto' ,        0.500  , ...
         'gradazione' ,           23  ) ;

C = struct( 'tipologia' ,       'chinotto' , ...
            'marca' , 'San_Pellegrino' , ...
          'contenuto' ,            0.500  , ...
         'gradazione' ,                0  ) ;

MC_mix = mix_bevande( M , C )
```

```
MC_mix =
  struct with fields:
     tipologia: 'amaro chinotto'
         marca: 'Montenegro San_Pellegrino'
     contenuto: 1
    gradazione: 11.5000
```

- Nel precedente esempio abbiamo definito due **struct** della stessa tipologia, cioè con gli stessi campi, per effettuare l'operazione di mix richiesta.
- Quando è richiesto di operare su un insieme di **struct** della stessa tipologia, cioè con gli stessi campi, possiamo nuovamente fare uso di array poichè i dati sono omogenei (**struct** con gli stessi campi).
- Si può definire direttamente un array di **struct**, con gli stessi campi, mediante le solite concatenazioni per riga/colonna:

```
>> s1 = struct( 'Nome' , 'Mario' , 'Voto' , 2 ) ;
>> s2 = struct( 'Nome' , 'Pippo' , 'Voto' , 8 ) ;
>> S = [ s1 , s2 ]
S =
  1×2 struct array with fields:
     Nome
     Voto
```

- Si può anche impiegare una definizione implicita di array di struct:

```
>> S(10) = struct( 'Nome' , 'Mario' , 'Voto' , 2 )
S =
  1×10 struct array with fields:
     Nome
     Voto
```

- In quest'ultimo caso si ha l'assegnazione dei valori dello **struct** solo nell'ultima posizione dell'array, mentre nelle posizioni precedenti si avrà un'inizializzazione con campi vuoti.

- Per definire un array di `struct` inizializzando direttamente anche i valori di tutti gli elementi, si può utilizzare la funzione `repmat`:

  `M = repmat( s , m1 , ... , mD )`

  che, nel caso che `s` sia uno `struct`, crea semplicemente un array di struct di dimensione `m1`$\times\cdots\times$`mD` dove ogni elemento è `s`:

```
>> s = struct( 'Nome' , 'Mario' , 'Voto' , 2 ) ;
>> S = repmat( s , 10 , 1 )
S =
  10×1 struct array with fields:
    Nome
    Voto
>> S(1)
ans =
  struct with fields:
    Nome: 'Mario'
    Voto: 2
```

- L'accesso e l'assegnazione negli array di struct avvengono esattamente come per l'accesso e l'assegnazione nelle matrici, cioè utilizzando uno o più indici (eventualmente vettorizzati), combinati con la notazione punto dello `struct` per accedere o assegnare un particolare campo:

```
>> s = struct( 'Nome' , 'Mario' , 'Voto' , 2 ) ;
>> S = repmat( s , 3 , 1 ) ; % S è inizializzato con Nome 'Mario' e Voto 2
>> S(1).Nome = 'Valeria' ;
>> S(1).Voto = 3 ;
```

- Nella parte 2 abbiamo visto l'uso della funzione `plot` per diagrammare curve e dati, senza assegnare nessuna variabile in output alla funzione:

```
plot( x , y ) ;
```

- La funzione `plot` può però restituire una variabile in output, se richiesta:

  `my_plot = plot(...)`

  dove `my_plot` è una variabile di un particolare tipo che contiene al suo interno tutte le informazioni necessarie a descrivere l'elemento grafico ottenuto:

```
>> my_plot
my_plot =
  Line with properties:

            Color: [0 0.4470 0.7410]
        LineStyle: '-'
        LineWidth: 0.5000
           Marker: 'none'
       MarkerSize: 6
  MarkerFaceColor: 'none'
            XData: 1
            YData: 0
            ZData: [1×0 double]

  Show all properties
>> class(my_plot)
ans =
    'matlab.graphics.chart.primitive.Line'
```

- Possiamo quindi pensare a questa variabile `my_plot` come un singolo *oggetto* descritto da varie *proprietà*, in maniera molto simile a quanto accade con gli `struct`. Si può operare sulle proprietà di questo oggetto con la stessa notazione utilizzata negli `struct` per accedere o assegnare il valore di un campo:

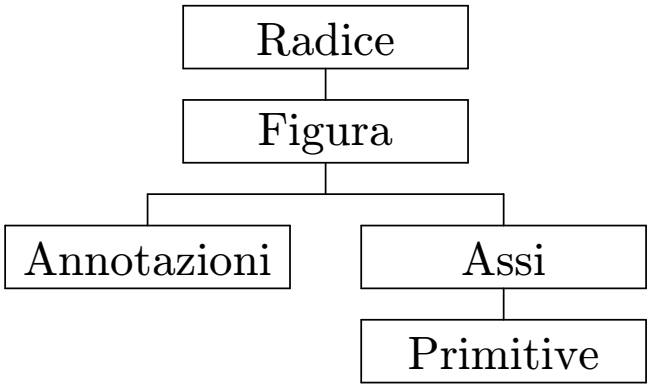
```
>> my_plot.Color
ans =
        0    0.4470    0.7410

>> my_plot.LineWidth = 1.5 ;
```

- Nell'ultimo caso, però, l'assegnazione del valore `1.5` alla proprietà `LineWidth` (spessore della linea grafica) non è statica ma provoca un immediato aggiornamento dell'elemento grafico descritto dalla variabile `my_plot`: questo accade perchè `my_plot` non è una semplice sequenza statica di dati, come nello `struct`, ma è un *oggetto* di una *classe*.
- In una classe non si definiscono solo le proprietà di un elemento appartenente a quella classe (un oggetto), ma anche delle funzioni (*metodi*) che permettono di operare sugli oggetti di quella classe. Possiamo ottenere la lista delle proprietà e dei metodi di una classe mediante le funzioni `properties` e `methods`:

```
>> properties( class(my_plot) ) ;
>> methods( class(my_plot) ) ;
```

- Più precisamente la variabile `my_plot` è un *handle* al relativo oggetto grafico, mediante il quale è quindi possibile operare sugli elementi grafici. In termini operativi possiamo confondere l'oggetto grafico con il relativo handle (la variabile `my_plot`).
- L'operazione di assegnazione di una proprietà di un oggetto grafico, per esempio usando la notazione punto, equivale a richiamare un metodo (`set`) che aggiorna quella particolare proprietà dell'oggetto grafico.
- Ogni elemento grafico in MATLAB è descritto da un oggetto grafico di una particolare classe, e tutti gli oggetti grafici sono organizzati secondo una struttura gerarchica ad albero:



  - A partire dalla radice (`groot`) è possibile esplorare tutti gli oggetti grafici mediante le proprietà `Children` e `Parent` di ogni oggetto.
  - Per cancellare un oggetto grafico `obj` si usa `delete(obj)`, che cancella anche tutti i suoi figli.
- Si può creare un oggetto grafico a qualsiasi livello della gerarchia: in tal caso vengono creati automaticamente di default tutti gli oggetti che nella gerarchia portano alla radice, sempre presente.

- Ad esempio, mediante **plot** si crea un oggetto grafico al livello più basso:

```
>> my_plot = plot( rand(10,1) ) ;
```

Scorrendo poi la gerarchia mediante **Children** (partendo dalla radice **groot**) o **Parent** (partendo da **my_plot**) si possono identificare gli oggetti grafici Figura e Assi che vengono automaticamente creati. Alternativamente, si possono usare le funzioni **gca** e **gcf** per ottenere gli handle degli Assi correnti o della Figura corrente:

```
>> assi   = my_plot.Parent
  ans =
    Axes with properties:
    XLim: [1 10]
    ...
>> figura = assi.Parent
  ans =
    Figure (1) with properties:
    Number: 1
    ...
```

```
>> assi = gca ;
>> figura = gcf ;
```

- Figure e Assi si possono creare esplicitamente mediante le funzioni **figure** e **axes**:

```
>> figura = figure(1) ;
>> assi   = axes() ;
```

- Abbiamo visto che l'oggetto grafico creato con **plot** appartiene agli oggetti primitivi della classe **matlab.graphics.chart.primitive.Line**. Le proprietà di più frequente utilizzo sono quelle inzialmente mostrate quando si richiama il relativo function handle, in particolare:
  - **Color**: definito mediante un vettore di 3 valori RGB normalizzati a 1;
  - **LineStyle**: **- , -- , : , -.** per linea continua, tratteggiata, puntinata o tratto-punto;
  - **LineWidth**: spessore linea;
  - **Marker**: tipo di marker tra **o,+,*,.,x,s,d,^,v,>,<,p,h**;
  - **MarkerSize**: dimensione marker;
- Oltre che con la notazione punto, le proprietà di un oggetto grafico si possono anche definire al momento della creazione dell'oggetto passandole come argomenti in ingresso alla funzione dopo gli argomenti principali, nell'ordine **'proprietà',valore**:

```
x = linspace( 0 , 1 , 100 ) ;
p = plot( x , x .^ 2 , 'Color' , [1 0 0] ) ;
p.LineWidth = 2 ;
```

- Gli assi sono oggetti grafici che servono a descrivere un sistema di riferimento cartesiano in 2 o 3 dimensioni e sono caratterizzati da un gran numero di proprietà. Vediamone le principali:
  - `Box`: booleano per attivare/disattivare le linee esterne che definiscono l'area di visualizzazione;
  - `DataAspectRatio`: vettore di 3 elementi che definisce il fattore di scala degli assi rispetto alla figura. Ad esempio, con `DataAspectRatio=[1 1 1]` un cerchio in 2D non appare deformato graficamente;
  - `FontName, FontSize`: per definire il tipo e la dimensione del font utilizzato per i numeri/testi visualizzati lungo gli assi;
  - `Position`: vettore di 4 elementi `[x0,y0,width,height]`, di default normalizzati a 1, che definiscono la posizione degli assi nella figura;
  - `TickLabelInterpreter`: interprete dei numeri/testi visualizzati lungo gli assi, può essere `'tex'` o `'latex'`;
  - `TickDir`: direzion dei ticks, può essere `'in'`, `'out'` o `'both'`;
  - `XLim, YLim, (ZLim)`: vettori di 2 valori che definiscono gli intervalli di visualizzazione lungo le 3 dimensioni;
  - `XScale, YScale, (ZScale)`: 'linear' o 'log' per scale lineari o logaritmiche;
  - `XGrid, YGrid, (ZGrid), XMinorGrid, YMinorGrid, (ZMinorGrid)`: booleani per attivare/disattivare la griglia in corrispondenza dei *ticks* e dei minorticks;
  - `Title, XLabel, YLabel, (ZLabel)`: oggetti grafici primitivi, quindi caratterizzati a loro volta da altre proprietà, per i testi da visualizzare come titolo e denominazione degli assi.

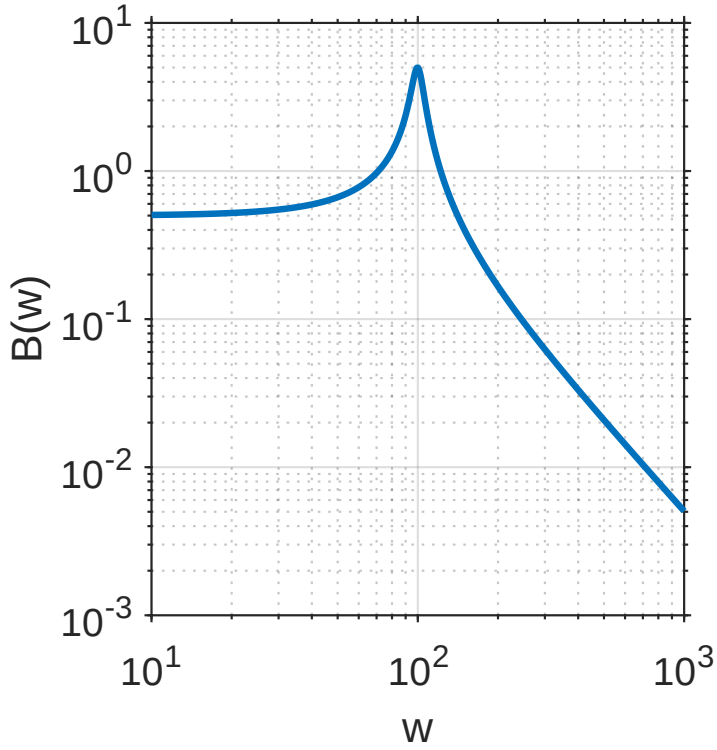
- Scrivere uno script che diagrammi la funzione

$$B(\omega) = \frac{A}{|k - m\omega^2 + j\omega c|}$$

nell'intervallo $[10^1, 10^3]$ utilizzando $A = 5000$, $k = 10000$, $m = 1$, $c = 10$. Utilizzare inoltre:

- scala logaritmica per entrambi gli assi;
- denominazione ω per l'ascissa e $B(\omega)$ per l'ordinata;
- posizionare esternamente i ticks;
- attivare la griglia principale e quella minore per tutti e due gli assi.

```
A = 5000 ;
k = 10000 ;
m = 1 ;
c = 10 ;
w = linspace( 10^1 , 10^3 , 1000 ) ;
B = A ./ abs( k - m*w .^ 2 + 1i*w*c ) ;
p = plot( w , B , 'LineWidth' , 1.5 ) ;
assi = gca ;
assi.XScale = 'log' ;
assi.YScale = 'log' ;
assi.XLabel.String = 'w' ;
assi.YLabel.String = 'B(w)' ;
assi.TickDir = 'out' ;
assi.XGrid = true ;
assi.YGrid = true ;
assi.XMinorGrid = true ;
assi.YMinorGrid = true ;
```



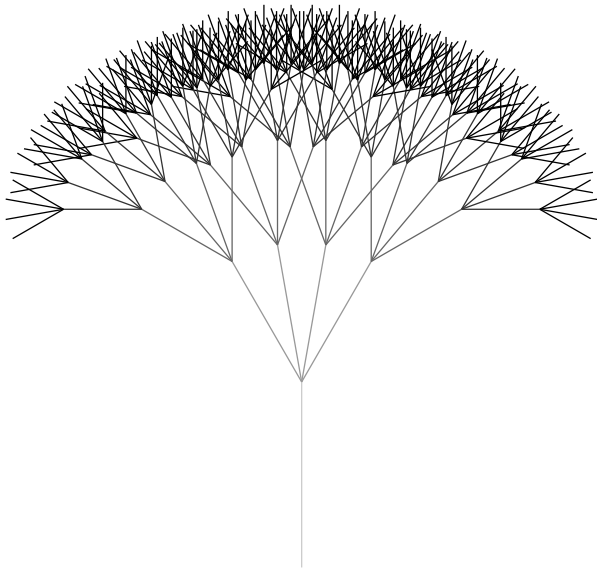
- *Albero ricorsivo.* Nel piano, un generico segmento (ramo) può essere identificato con due punti $(z_0, z_1 = z_0 + r_0)$ nel piano complesso. Un albero si può costruire partendo da un ramo iniziale ed aggiungendo all'estremità z_1 un certo numero N_{rami} di nuovi rami, ripetendo poi ricorsivamente l'operazione di aggiunta per ogni nuovo ramo e fermandosi ad un certo punto. Ognuno dei nuovi rami sarà a sua volta un segmento $(z_1, z_1 + r_1)$: scegliamo di ottenere la nuova direzione r_1 ruotando di un angolo α e scalando di un fattore $\rho < 1$ la direzione r_0 del ramo padre. In termini di numeri complessi questa operazione è data dal prodotto $r_1 = r_0 c$ dove $c = \rho e^{j\alpha} = \rho(\cos\alpha + j \sin\alpha)$:

ramo.m

```
% Input: z0, punto iniziale del ramo
%        r0, direzione del ramo (punto finale = z0+r0)
%        c, vettore di lunghezza N_rami dei fattori moltiplicativi
%           rho e^( j alpha ) per ogni sottoramo
%        livello, intero del livello del ramo considerato
%        livello_max, massimo numero di livelli
function ramo( z0 , r0 , c , livello , livello_max )
  z1 = z0 + r0 ;
  plot( [z0 z1] ) ;
  if livello < livello_max
    for sottoramo = 1 : length(c)
      r1 = r0 * c(sottoramo) ;
      ramo( z1 , r1 , c , livello+1 , livello_max ) ;
    end
  end
end
```

- Per semplicità, per ognuno degli N_{rami} sottorami di ogni ramo scegliamo degli angoli di rotazione α uniformemente distribuiti nell'intervallo $[-\alpha_M, \alpha_M]$:

```
fig = figure( 1 ) ;
hold on ;
alfa_M = 30 * ( pi / 180 ) ;
N_rami = 4 ;
alfa   = alfa_M * linspace(-1, 1, N_rami) ;
rho    = 0.75 ;
c      = rho * exp( 1i * alfa ) ;
z0 = 0 ;
r0 = 1i ;
livelli = 6 ;
ramo( z0 , 1i , c , 1 , livelli ) ;
axis equal ;
axis off ;
```



- Attenzione a non esagerare con il numero massimo di livelli L e/o con N_{rami}: il numero totale R dei rami disegnati è

$$R = 1 + N_{rami} + N_{rami}^2 + \cdots + N_{rami}^{L-1} = \frac{N_{rami}^L - 1}{N_{rami} - 1}$$

e può essere verificato a posteriori accedendo alla proprietà `Children` dell'oggetto assi:

```
>> assi = gca ;
>> length( assi.Children )
    1365 % = ( 4^6 - 1 ) / ( 4 - 1 )
```

- Se il numero di livelli è molto grande, l'altezza H dell'albero è data da:

$$H = H_0 + H_0\rho + H_0\rho^2 + \cdots = \frac{H_0}{1-\rho}$$

  se $\rho < 1$, dove H_0 è la lunghezza del primo ramo.
- Se il numero di livelli è molto grande, la somma E delle lunghezze di tutti i rami è:

$$E = H_0\left(1 + \rho N_{rami} + (\rho N_{rami})^2 + \cdots\right) = \frac{H_0}{1-\rho N_{rami}}$$

  se $\rho N_{rami} < 1$.
- Può quindi accadere di avere un albero di altezza finita ($\rho < 1$) ma lunghezza totale dei rami infinita se $N_{rami} \geq 1/\rho$.

- Quando si ha la necessità di visualizzare grandezze che dipendono da 2 variabili, come matrici ottenute dal calcolo di una funzione $f(x, y)$ di due variabili, è molto comodo utilizzare la funzione **meshgrid** per generare delle matrici per ciascuna delle due variabili:

  ```
  [ x , y ] = meshgrid( vx , vy )
  ```

  dove **vx** è un vettore di n valori della prima variabile (x) e **vy** è un vettore di m valori della seconda variabile (y). **x** ed **y** sono matrici $m \times n$ che definiscono il prodotto cartesiano dei due vettori **vx** e **vy**, contenenti perciò tutte le possibili combinazioni dei valori dei due vettori:

```
>> vx = 0 : 4 ;
>> vy = 100 : 101 ;
>> [ x , y ] = meshgrid( vx , vy )
x =
     0     1     2     3     4
     0     1     2     3     4
y =
   100   100   100   100   100
   101   101   101   101   101
```

- Il prodotto cartesiano tra due vettori si può anche ottenere con le seguenti operazioni vettoriali già viste:

```
vx = 0 : 4 ;
vy = (100 : 101)' ;
x = 0*vy +   vx ;
y =   vy + 0*vx ;
```

- Avendo definito le due matrici `x` ed `y` come prodotto cartesiano dei due vettori `vx` e `vy`, una funzione di due variabili può essere espressa in termini di semplici operazioni matriciali già viste, ricordando sempre il punto nelle operazioni di prodotto/divisione/elevamento a potenza. Ad esempio la funzione $f(x, y) = \cos(xy)$ potrà essere implementata con:

```
f = @(x,y) cos( x .* y )
```

  dove `.*` è quindi l'operazione di prodotto elemento per elemento tra le due matrici prodotto cartesiano `x` ed `y`, che hanno quindi stessa dimensione, mentre la funzione `cos` è nativamente implementata per operare con ingressi matriciali.
- Quando una funzione $f(x, y)$ è definita algebricamente per mezzo di operazioni tra matrici, come ad esempio $f(\mathbf{x}) = \mathbf{v}^T\mathbf{x}$ con $\mathbf{v}$ vettore 2×1, oppure $f(\mathbf{x}) = \mathbf{x}^T\mathbf{A}\mathbf{x}$ con $\mathbf{A}$ matrice (simmetrica) 2×2, bisogna esplicitarla in termini di x ed y:

$$f(\mathbf{x}) = \mathbf{v}^T\mathbf{x} = v_1 x + v_2 y$$

```
f(x,y) = @(x,y) v(1)*x + v(2)*y ;
```

$$f(\mathbf{x}) = \mathbf{x}^T\mathbf{A}\mathbf{x} = a_{11}x^2 + 2a_{12}xy + a_{22}y^2$$

```
f(x,y) = @(x,y) a(1,1)*x.^2 + 2*a(1,2)*x.*y + a(2,2)*y.^2 ;
```

- Le curve di livello di una funzione di due variabili $f(x, y)$ sono date dall'insieme di valori (x, y) tale per cui $f(x, y) = const$. Per visualizzare queste curve si può utilizzare `fcontour` che opera direttamente sul function handle di $f(x, y)$:

  ```
  fc = fcontour( f , intervalli )
  ```

  dove `f` è il function handle della funzione di due variabili implementata come visto in precedenza e `intervalli` è il vettore `[x1 x2 y1 y2]` che definisce gli intervalli $[x1, x2]$ lungo x e $[y1, y2]$ lungo y entro i quali operare.
- Alternativamente, si può utilizzare `contour` che opera su una generica matrice `z=f(x,y)` invece che direttamente sul function handle:

  ```
  [ M , fc ] = contour( x , y , z , livelli )
  ```

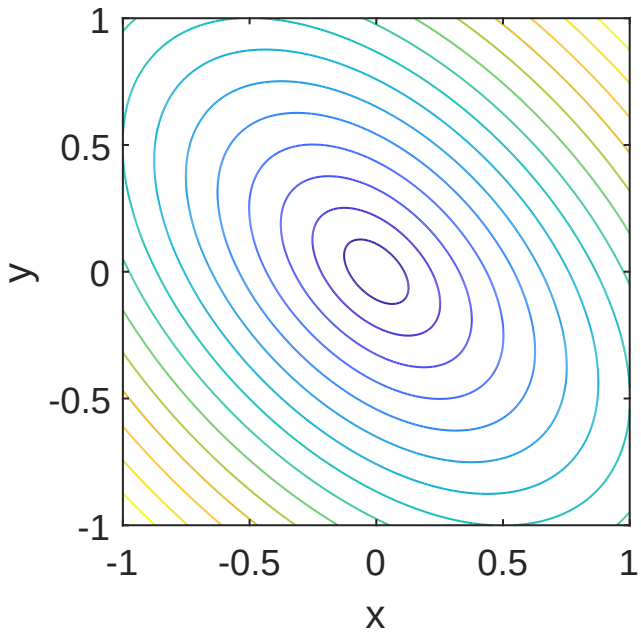
  dove `livelli` è il numero di livelli che si vuole visualizzare oppure il vettore dei valori dei livelli voluti e `M` contiene i punti delle curve di livello. Operando su una matrice e non su un function handle, `contour` è applicabile soprattutto nei casi dove la funzione di cui si vogliono visualizzare le curve di livello è nota solo nei punti `(x,y)`.
- In entrambi i casi i valori dei livelli di ciascuna curva possono essere anche definiti mediante le proprietà `'LevelList'` o `'LevelStep'` dell'oggetto grafico al momento della chiamata, oppure anche successivamente attraverso l'handle `fc`.

- Diagrammare le curve di livello della funzione

$$f(x, y) = \sqrt{x^2 + xy + y^2}$$

  in $[-1, 1] \times [-1, 1]$ utilizzando 15 curve di livello.
- Conviene utilizzare **contour** per definire direttamente il numero di curve di livello:

```
f = @(x,y) sqrt( x.^2 + x.*y + y.^2 ) ;
n = 500 ;
vx = linspace(-1, 1, n) ;
[x, y] = meshgrid(vx, vx) ;
z = f(x,y) ;
livelli = 15 ;
contour(x, y, z, livelli) ;
axis equal ;
xlabel('x') ;
ylabel('y') ;
```



- Dall'osservazione delle curve di livello si possono desumere due direzioni a $\pm 45°$, confermate dal fatto che $x^2 + xy + y^2 = \frac{3}{4}(x + y)^2 + \frac{1}{4}(x - y)^2$ dove $x + y = (x, y) \cdot (1, 1)$ e $x - y = (x, y) \cdot (1, -1)$ sono le due direzioni a $\pm 45°$.

- Il grafico $z = f(x, y)$ di una funzione di due variabili è quindi una superficie. Per visualizzare una superficie in tale forma si può utilizzare `fsurf` che opera direttamente sul function handle di $f(x, y)$:

  ```
  fs = fsurf( f , intervalli )
  ```

  dove `f` è il function handle della funzione di due variabili implementata come visto in precedenza e `intervalli` è il vettore `[x1 x2 y1 y2]` che definisce gli intervalli $[x1, x2]$ lungo x e $[y1, y2]$ lungo y entro i quali operare.
- Alternativamente, si può utilizzare `surf` che opera su una generica matrice `z=f(x,y)` invece che direttamente sul function handle:

  ```
  fs = surf( x , y , z )
  ```

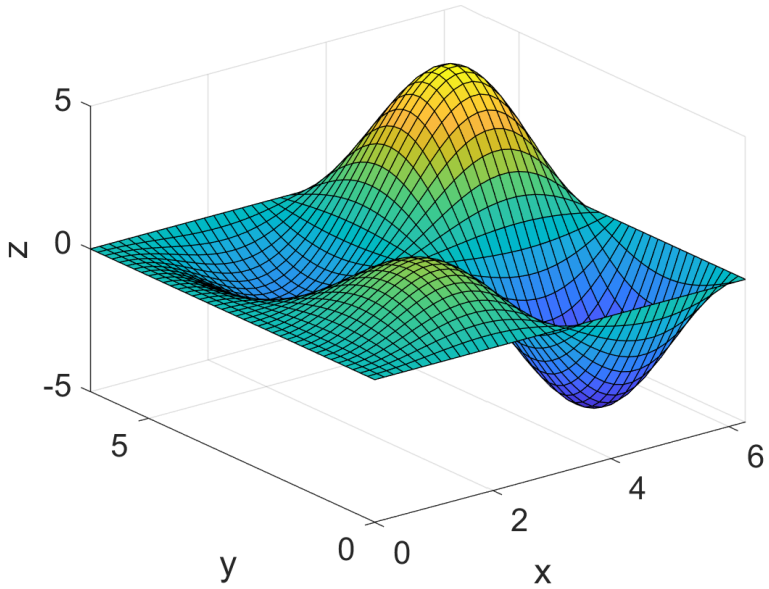
  Analogamente al caso delle curve di livello, operando su una matrice invece che su un function handle, `surf` è applicabile nei casi dove la funzione di cui si vuole diagrammare il grafico è nota solo nei punti `(x,y)`.
- In entrambi i casi si può personalizzare l'aspetto della superficie definendo le proprietà del relativo oggetto grafico sia al momento della chiamata che successivamente attraverso l'handle `fs`.

- Diagrammare il grafico $z = f(x, y)$ della funzione

$$f(x, y) = x \sin x \sin(y)$$

in $[0, 2\pi] \times [0, 2\pi]$.

```
f = @(x,y) x .* sin(x) .* sin(y) ;
n = 40 ;
vx = linspace(0, 2*pi, n) ;
[x, y] = meshgrid(vx, vx) ;
z = f(x,y) ;
fs = surf(x, y, z) ;
xlabel('x') ;
ylabel('y') ;
zlabel('z') ;
```



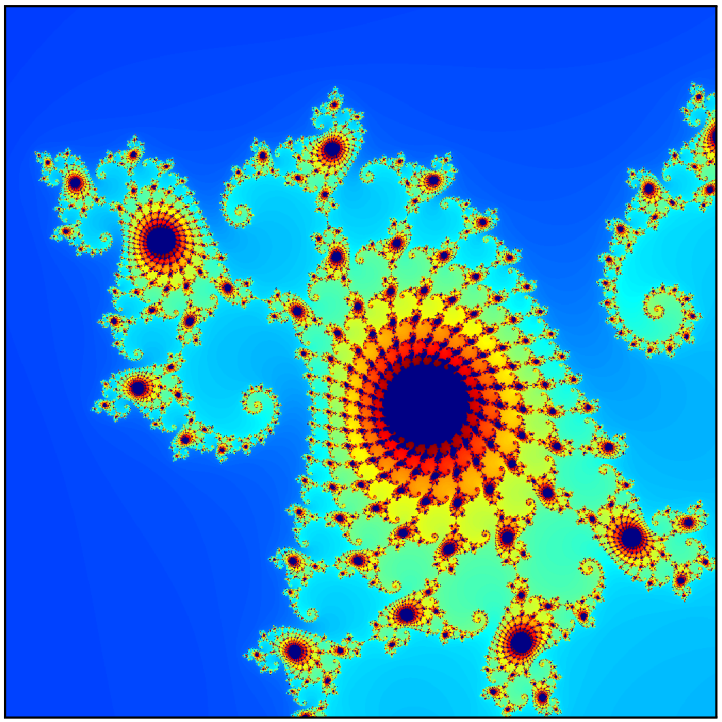
- È quindi poi possibile personalizzare l'aspetto della superficie agendo sul relativo handle `fs`. Per esempio, si possono eliminare le linee di superficie che definiscono la griglia cartesiana, impostare un colore unico per la superficie ed aggiungere un oggetto grafico d'illuminazione `light` per conferire un aspetto 3D:

```
fs.EdgeColor = 'none' ;
fs.FaceColor = [0 .8 0] ;
fs.FaceLighting = 'gouraud' ;
light ;
```

- Implementando la seguente iterazione

$$z_{i+1} = z_i^2 + z_0 \qquad i = 0, \ldots, i_{max}$$

per i punti $z_0 = x + jy$ del piano complesso con $x \in v_x$ e $y \in v_y$ e tenendo traccia dell'iterazione $i = i_P$ per la quale $|z_{i+1}| > 2$ si ottiene:



```
x0 = -0.748766700 ;
y0 =  0.123640855 ;
d  =  1.5e-8 ;

n  = 1000 ;
vx = linspace(x0-d, x0+d, n) ;
vy = linspace(y0-d, y0+d, n) ;
[x, y] = meshgrid(vx, vy) ;
z0 = x + 1i*y ;
zi = z0 ;
iP = z0*0 ;

i_max = 200 ;
for i = 0 : i_max
  zi = zi .^ 2 + z0 ;
  ix = abs(zi) > 2 ;
  z0(ix) = 0 ;
  zi(ix) = 0 ;
  iP(ix) = i ;
end

image(vx, vy, iP) ;
axis equal ;
assi = gca ;
assi.Colormap = jet(i_max) ;
```

- Salvataggio o lettura di file `.fig`. Per l'I/O da/verso file di oggetti grafici di tipo figura, quindi solo interpretabili con MATLAB, si possono utilizzare le seguenti funzioni:

```
savefig(fig_h, filename)
fig_h = openfig(filename)
```

  dove `fig_h` è l'handle della figura in oggetto e `filename` segue le convenzioni già viste per quanto riguarda l'indirizzo assoluto o relativo alla cartella di lavoro. Tutti gli oggetti grafici presenti nella figura vengono salvati/letti, ossia tutti gli oggetti grafici figli di `fig_h`.
- Per l'esportazione di file grafici intepretabili esternamente a MATLAB, si usa la seguente funzione:

```
exportgraphics(ogg, filename, opzioni)
```

  dove `ogg` è l'oggetto grafico che si vuole esportare ed il formato di file grafico voluto va indicato nel `filename`. I formati utilizzabili sono:
  - immagini raster: `png`, `jpg/jpeg`, `tif/tiff`;
  - immagini vettoriali: `pdf`, `eps`, `emf`.
- Le `opzioni`, definite nella sequenza `'nome_opzione',valore` riguardano:
  - `'Resolution'` per definire la risoluzione in DPI (dots per inch) nelle immagini raster;
  - `'ContentType'` per forzare un contenuto vettoriale (`'vector'`) o raster (`'image'`) nei formati vettoriali.

- Esportazione di un grafico in formato vettoriale .pdf:

```
x = linspace(0, 2*pi, 100) ;
y = sin(x) ;
plot(x, y) ;
xlabel('x') ;
ylabel('y') ;
assi = gca ;
exportgraphics(assi, 'Grafico_sin.pdf') ;
```

- Esportazione di un grafico in formato raster .png specificando una risoluzione di 300 DPI:

```
x = linspace(0, 2*pi, 100) ;
y = sin(x) ;
plot(x, y) ;
xlabel('x') ;
ylabel('y') ;
assi = gca ;
exportgraphics(assi, 'Grafico_sin.png', 'Resolution', 300) ;
```