Laboratorio di Informatica T

Grafici Cartesiani con Matlab

Grafici Con Matlab

Matlab permette anche di disegnare facilmente dei grafici

La prima cosa da fare è costruire una nuova "figura":

```
figure()
```

- La funzione **`figure`** apre una nuova finestra...
- ...In cui verrà inserito il disegno

In molti casi, questo passaggio può essere saltato

- Se non è stata ancora costruita una figura...
- ...Molte funzioni di disegno se ne accorgono...
- ...E chiamano **`figure`** automaticamente

Grafici Con Matlab

Supponiamo di volere costruire un grafico cartesiano

Per prima cosa dobbiamo costruire il vettore delle x

Possiamo usare **`linspace`**:

```
x = linspace(-2*pi, 2*pi, 200)
```

- Provandolo, noterete che il vettore restituito contiene molti elementi
- Il fatto che venga visualizzato in automatico è scomodo

La visualizzazione può essere disabilitata aggiungendo un ";"

Quindi basta scrivere:

```
x = linspace(-2*pi, 2*pi, 200);
```

Plotting

Ora otteniamo il vettore con i valori per l'asse delle `y`

Calcoliamo per esempio la funzione "seno"

- Basta applicarla al vettore **`x`**...
- ...Perché **`sin`** opera elemento per elemento

```
y = sin(x);
```

In questo modo otteniamo:

- I valori della funzione **`sin`**...
- ...Corrispondenti agli elementi del vettore **`x`**

Plotting

A questo punto possiamo disegnare il grafico

Si utilizza la funzione **`plot`**

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

Avremmo anche potuto scrivere direttamente:

```
plot(x, sin(x)) % senza usare una variabile per y
```

Si può anche specificare un colore:

```
plot(x, sin(x), 'b') % b = blue
```

- Guardate la documentazione di **`plot`** per altri dettagli!

Plotting

Si può aggiungere una griglia con:

```
grid()
```

Per disegnare più curve sovrapposte:

```
figure()                % Nuova figura
plot(x, sin(x), 'b')    % Prima curva, in blu
hold on                 % Attiva la modalità "hold"
plot(x, cos(x), 'g')    % Seconda curva, in verde
hold off                % Disattiva la modalità "hold"
```

- Senza la modalità **`hold`** ogni plot rimpiazza il precedente

Grafici Cartesiani

Disegnate, per $x \in [-4, 4]$ le seguenti funzioni:

$$(1) \quad x^2 - x \qquad (2) \quad \frac{1}{1 + |x|}$$

$$(3) \quad \frac{1}{1 + e^{-x}} \qquad (4) \quad \frac{1}{x|x|}$$

Prima di disegnarle, cercate di intuire se sono:

- Continue
- Derivabili (senza "spigoli")
- Concave/convesse (eventualmente)

E quindi verificatelo visivamente

Grafici Cartesiani (Soluzione)

Soluzione:

```
x = linspace(-4, 4, 200);   % Notate il ;
plot(x, x.^2 - x)           % Cont., derivabile, conv.
plot(x, 1./(1 + abs(x)))    % Cont., non derivabile
plot(x, 1./(1 + exp(-x)))   % Cont., derivabile
plot(x, 1./(x.*abs(x)))     % Non cont., non derivabile
```

- Notate che per l'ultima funzione il grafico ha un artefatto!
- La funzione non è continua, ma si vede comunque una linea
- Succede perché in `0` la funzione non è definita...
- ...ma il punto subito prima e subito dopo sono ben definiti...
- ...quindi **`plot`** li congiunge con una linea

Laboratorio di Informatica T

Istruzioni di Iterazione: Cicli `for`

Cosa ci Serve per Programmare?

Il nostro obiettivo è formulare algoritmi, giusto?

Proviamo a capire se possiamo farlo con i costrutti visti finora

Somma degli Elementi di un Vettore

Proviamo a calcolare la somma degli elementi di un vettore

Come fare?

- La somma è inizialmente uguale a 0
- Consideriamo tutti gli elementi successivi
- Aggiungiamoli uno ad uno alla somma

Abbiamo gli strumenti per codificarlo?

Tornando alla Somma di un Vettore...

Vediamo come usare queste informazioni per il nostro algoritmo

- Supponiamo che il nostro vettore si chiami `v`

```
v = % il nostro vettore
s = 0;
<facciamo variare una variabile ii da 1 a length(v)>
  s = s + v(ii)
```

- Quando `ii` avrà raggiunto il valore `length(v)`...
- ...la variabile `s` conterrà la somma

Ci rimane da capire come gestire l'indice `ii`

- **Nota:** È utile terminare con ";" le istruzioni intermedie
- **Eccezione:** se ci sono problemi, è bene stampare tutto

Istruzioni di Iterazione: Ciclo `for`

Matlab ci permette di farlo utilizzando una particolare istruzione

Una istruzione è una notazione che, quando viene eseguita, ordina all'elaboratore di fare qualcosa

La definizione è un po' vaga...

- Le istruzioni sono i "passi elementari" del programma
- Un programma è in prima battuta una sequenza di istruzioni
- Un esempio: espressione + [INVIO] = una istruzione

Istruzioni di Iterazione: Ciclo `for`

Matlab ci permette di farlo utilizzando una particolare istruzione

Una istruzione è una notazione che, quando viene eseguita, ordina all'elaboratore di fare qualcosa

Ora vediamo un nuovo tipo di istruzioni:

Istruzioni di iterazione o "cicli": permettono di eseguire ripetutamente una porzione di codice

Istruzioni di Iterazione: Ciclo `for`

La nostra prima istruzione di iterazione si chiama "ciclo `for`"

Sintassi:

```
for <variabile> = <vettore>
  <corpo>
end % o anche endfor
```

- **`<corpo>`** è una sequenza di istruzioni
- Il corpo viene ripetuto per ogni elemento di **`<vettore>`**

Ad ogni ripetizione (i.e. iterazione):

- **`<variabile>`** assume il valore di un elemento di **`<vettore>`**

Nota: il simbolo "=" in questo caso non ha il solito significato...

Istruzioni di Iterazione: Ciclo `for`

Un esempio semplice:

```
for v = [2, 4, 6]
  v % Risp: ans = 2 (non c'è il ";")
    %       ans = 4
    %       ans = 6
end
```

Quando inserite una istruzione `for` nella finestra dei comandi:

- Matlab non esegue subito l'istruzione
- Vi permette di continuare a scrivere e premere [INVIO]
- Finché non inserite la parola chiave **`end`**

Solo allora l'istruzione **`for`** è completa (e quindi eseguibile)

Ciclo `for` e Somma di un Vettore

Il ciclo `for` ci permette di sommare gli elementi di `v`:

```
s = 0;
for ii = 1:length(v)
  s = s + v(ii)
end
```

Oppure, in modo ancora più semplice:

```
s = 0;
for w = v % w assume uno per uno i valori di v
  s = s + w
end
```

La funzione predefinita **`sum(<vettore>)`** fa (più o meno) questo

Somma di una Matrice

Alziamo un po' il livello di difficoltà

Supponiamo di voler sommare gli elementi di una matrice

Un possibile algoritmo:

```
M = % la nostra matrice
s = 0
<per ogni elemento della matrice>
  s = s + <elemento>
```

Come iterare sugli elementi di una matrice?

- Potremmo provare con un ciclo `for`...
- ...del resto, ha funzionato nel caso precedente

Somma di una Matrice

Vediamo cosa succede iterando con un `for` su una matrice:

```
M = [1, 2, 3; 4, 5, 6; 7, 8, 9]; % [1, 2, 3;
                                  %  4, 5, 6;
                                  %  7, 8, 9]

for w = M
  w % giusto per stampare w
end
```

La variabile `w` assume i valori:

- `[1; 4; 7]`, poi `[2; 5; 8]`, poi `[3; 6; 9]`

Ossia le colonne di `M`!

Come possiamo fare per iterare sugli elementi?

Somma di una Matrice

Una soluzione semplice: usare due cicli for

```
M = [1, 2, 3; 4, 5, 6; 7, 8, 9];
for C = M     % itero sulle colonne
  for v = C   % poi sugli elementi della colonna
    v
  end % fine ciclo interno
end % fine ciclo esterno
```

- `for` è una istruzione...
- ...Quindi il corpo di un `for` può contenere un altro `for`!

I due cicli si dicono innestati

- `end` si riferisce sempre all'ultimo `for` non ancora chiuso

Somma di una Matrice

Possiamo anche fare iterare due indici:

```
M = % la nostra matrice
s = 0
[n, m] = size() % Recupero le due dimensioni
for ii = 1:n    % Itero sulle righe
  for jj = 1:m  % Itero sulle colonne
    s = s + M(ii, jj)
  end
end
```

- `size(M)` restituisce un vettore `[<nrighe>, <ncolonne>]`...
- Ma se viene chiamato con due variabili a sx di `=`...
- ...Restituisce due valori distinti

Riprenderemo questo comportamento nella prossima lezione

Somma di una Matrice

Possiamo anche fare iterare due indici:

```
M = % la nostra matrice
s = 0
[n, m] = size() % Recupero le due dimensioni
for ii = 1:n    % Itero sulle righe
  for jj = 1:m  % Itero sulle colonne
    s = s + M(ii, jj)
  end
end
```

NOTA: vedete che il corpo di ogni `for` è leggermente indentato?

- Non è obbligatorio, ma rende il codice più leggibile
- Consiglio: mantenete sempre il codice ben indentato!

Somma di una Matrice

In alternativa, possiamo usare la funzione `sum`

Ci vuole però qualche accortezza:

```
M = [1, 2; 3, 4] % [1, 2]
                 % [3, 4]
sum(M) % denota [4, 6]
```

- Se applicata ad una matrice...
- ...`sum` calcola la somma colonna per colonna

Per avere la somma di tutti gli elementi:

```
sum(sum(M)) % denota 10
```

- La seconda applicazione di `sum` ottiene la somma totale

Laboratorio di Informatica T

Istruzioni Condizionali

Massimo di un Vettore

Supponiamo di voler calcolare il massimo di un vettore

Cosa riusciamo a scrivere con le istruzioni che conosciamo?

```
V = % il nostro vettore
y = V(1) % primo tentativo: max = il primo elemento
for ii = 2:length(V) % iteriamo sul resto del vettore
  <se V(ii) è maggiore di y, allora y = V(ii)>
end
```

Abbiamo bisogno di un costrutto che:

- Permetta di stabilire se `V(ii)` è maggiore di `y`...
- ...Se questo è vero, esegua l'istruzione `y = V(ii)`

Questa funzionalità è fornita dalle istruzioni condizionali

Istruzione Condizionale `if`

Noi utilizzeremo una sola istruzione condizionale, chiamata `if`

Sintassi:

```
if <espressione>
  <corpo1>
else        % opzionale
  <corpo2> % opzionale
end
```

- **`<espressione>`** viene valutata ed interpretata come valore logico
- **`<corpo1>`** e **`<corpo2>`** sono sequenze di istruzioni
- **`<corpo1>`** esegue se **`<espressione>`** denota **`true`** (o $\neq 0$)
- **`<corpo2>`** esegue se **`<espressione>`** denota **`false`** (o 0)

Intruzione `if` nel Nostro Algoritmo

Grazie all'istruzione `if` possiamo formulare il nostro algoritmo:

```
V = % il nostro vettore
y = V(1)
for ii = 2:length(V)
  if V(ii) > y
    y = V(ii)
  end
end
```

- L'indentazione non è necessaria, ma è fortemente consigliata

Ricordate:

- Se un programma è ben leggibile...
- ...È più difficile fare errori!

Condizioni Complesse

Possimo esprimere condizioni complesse con gli operatori logici

Per esempio:

```
if (x >= 3) & (x < 5) % true se x è tra 3 e 5
  <corpo>
end
```

E se dobbiamo verificare una condizione su un intero vettore?

- E.g. verificare se `v` contiene almeno uno `0`
- Una prima soluzione (non pratica)

```
(V(1) == 0) | (V(2) == 1) | (V(3) == 0) ...
```

Funzioni `all` e `any`

Possiamo usare due funzioni predefinite:

```
all(<vettore>) % true se tutti gli elementi sono true
any(<vettore>) % true se almeno un elemento è true
```

- In pratica, sono un grande "and" e un grande "or"

Qualche esempio:

```
a = [1, 2, 3];
all(a < 3) % all([true, true, false]) --> false
any(a < 2) % any([true, false, false]) --> true
```

- Se applicate a matrici, **`all`** ed **`any`** operano colonna per colonna
- ...Esattamente come **`sum`**

Laboratorio di Informatica T

Ciclo `while` e Teorema del Programma Strutturato

Funzione Zeta di Riemann

Supponiamo di voler calcolare la somma della serie (con $s > 1$):

$$\zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s}$$

- Si tratta della funzione Zeta di Riemann (per num. reali)

Funzione Zeta di Riemann

Supponiamo di voler calcolare la somma della serie (con $s > 1$):

$$\zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s}$$

- Si tratta della funzione Zeta di Riemann (per num. reali)

Proviamo ad abbozzare un algoritmo:

```
z = 0;
for n = 1:???? % Il problema è qui
  z = 1 / n.^s
end
```

- La serie ha infiniti termini!

Funzione Zeta di Riemann

Supponiamo di voler calcolare la somma della serie (con $s > 1$):

$$\zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s}$$

- La funzione non è computabile in Matlab
- Però possiamo approssimarla!

Tipicamente: ci si ferma ad un certo livello di precisione

- Sia $\zeta^{(k)}(s)$ la nostra approssimazione dopo k iterazioni
- Ci possiamo fermare quando $\left|\zeta^{(k)}(s) - \zeta^{(k-1)}(s)\right| < \varepsilon$
- Dove ε è la tolleranza che riteniamo accettabile

Però ancora non sappiamo a priori il numero di iterazioni...

Ciclo `while`

Ci serve una istruzione di iterazione capace di:

- Fermarsi in base al soddisfacimento di una condizione...
- ...Anziché alla fine di un vettore

Una istruzione di questo tipo è il ciclo `while`

Sintassi:

```
while <espressione> % vera o falsa
  <corpo>
end
```

- Il corpo viene ripetuto...
- ...Fintanto che `<espressione>` denota `true` (o $\neq 0$)

Funzione Zeta di Riemann

Possiamo usare **`while`** per scrivere un algoritmo per $\zeta(s)$:

$$\zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s}$$

```
z = 0;                              % val. della somma
n = 1;
old_z = -Inf;                       % vecchio z
while abs(z - old_z) > 1e-6 % 1e-6 è la tolleranza
  old_z = z;                        % memorizzo il vecchio z
  z = z + 1 ./ n.^s;
  n = n + 1;                        % incremento n
end
```

- `Inf` è un valore speciale che denota ∞

Cicli Infiniti

Il ciclo `while` non fornisce garanzie di terminazione

Per esempio:

```
n = 1;
s = 0;
while n < 10
   s = s + n;
end
```

- Questo ciclo non termina
- Perché `n` non viene incrementato!

Se vi capita, niente panico: basta premere **[CTRL+C]**

Istruzione `break`

In alcuni casi può essere utile interrompere un ciclo

Lo si può fare con l'istruzione **`break`**. Per esempio

```
...
while true      % Di base, non smetto mai di iterare
  old_z = z;
  z = z + 1 ./ n.^s;
  n = n + 1;
  if abs(z - old_z) < tol
    break
  end
end
```

- Qui break è usato per controllare la condizione in fondo al ciclo

Istruzione `break`

Altro esempio: controllare se un vettore `v` contiene `0`

Istruzione `break`

Altro esempio: controllare se un vettore `v` contiene `0`

- Un primo metodo (usando funzioni predefinite):

```
any(V == 0)
```

Istruzione `break`

Altro esempio: controllare se un vettore `v` contiene `0`

- Un primo metodo (usando funzioni predefinite):

```
any(V == 0)
```

- Un secondo metodo (senza funzioni predefinite):

```
trovato = false;
for w = V
  if w == 0
    trovato = true;
  end
end
```

Istruzione `break`

Possiamo migliorare l'algoritmo:

- Appena troviamo `0`, non serve controllare il resto del vettore!
- Possiamo interrompere subito il ciclo, con **`break`**

```
trovato = false;
for w = V
  if w == 0
    trovato = true;
    break; % interrompe il ciclo corrente
  end
end
```

- L'istruzione **`break`** interrompe il ciclo in cui essa compare

Programmazione Strutturata

Le istruzioni condizionali e di iterazione:

- Sono anche note come istruzioni di controllo di flusso
- Sono importanti per un particolare risultato:

Teorema del Programma Strutturato: la possibilità di comporre istruzioni per sequenza, condizione ed iterazione è sufficiente a codificare qualunque algoritmo

- Composizione per sequenza = scrivere le istruzioni una dopo l'altra

Quasi tutti i linguaggi si basano su questi tre metodi di composizione

Laboratorio di Informatica T

Esercizio: Prodotto di un Vettore

Esercizio: Prodotto di un Vettore

- Se non l'avete già fatto, create una cartella per questa lezione
- Create un file di scrip **`my_prod.m`**

Quindi:

- Definite un vettore **`x`** (non importa quale sia il contenuto)
- Calcolate la somma degli elementi del vettore

Matlab permette di farlo con:

```
prod(X)
```

- Confrontate i vostri risultati con quelli di **`prod`**

Esercizio: Prodotto di un Vettore

Per ottenere velocemente dei dati di test potete usare:

```
rand(N)      % Matrice quadrata
rand(M, N)   % Matrice MxN
```

La funzione **`rand`**

- Restituisce matrici di numeri (pseudo) casuali in $]0, 1[$
- Ha la stessa interfaccia di **`zeros`**, **`ones`**, etc.

Analogamente, le funzioni:

```
randi(MAX, N)      % Matrice quadrata
randi(MAX, M, N)   % Matrice MxN
```

- Fanno lo stesso, ma con numeri interi tra `1` e **`MAX`**

Laboratorio di Informatica T

Esercizio: Max

Esercizio: Max

Matlab permette di trovare il massimo elemento di un vettore con:

```
max(X)
```

Nel file di script funzione **`my_max`**:

- Definite un vettore **`X`**, da utilizzare come esempio
- Calcolare il massimo degli elementi di **`X`**

Confrontate il vostro risultato con quello di **`max`** di Matlab

- Potere sempre usare **`rand`**/**`randi`** per ottenere dei vettori di test

Laboratorio di Informatica T

Esercizio: Indicizzazione
con Vettore di Indici

Esercizio: Indicizzazione con Vettore di Indici

Abbiamo visto che Matlab permette di accedere ad un vettore con:

```
V(I)
```

- `V` è un vettore
- `I` è un vettore di indici

Nel file di script **`my_index.m`**:

- Definite un vettore `X` di dati ed un vettore `I` di indici
- Calcolate il vettore `Y` con gli elementi di `X`...
- ...Alle posizioni specificate da `I`

Confrontate il vostro risultato con quello di Matlab

- Potere sempre usare **`rand`**/**`randi`** per ottenere dei vettori di test

Elementi di Informatica e Applicazioni Numeriche T

Esercizio: Somma di Matrici

Esercizio: Somma di Matrici

In un file di script **`my_msum`**:

- Definite due matrici **`A`** e **`B`** di esempio, con la stessa dimensione
- Calcolatene la somma

Confrontate il vostro risultato con quello dell'operatore **`+`** di Matlab

- Potere sempre usare **`rand`**/**`randi`** per ottenere le matrici di test

Laboratorio di Informatica T

Esercizio: Prodotto Scalare

Esercizio: Prodotto Scalare

Matlab permette di effettuare il prodotto scalare con:

```
X * Y' % Con X e Y vettori riga
dot(X, Y) % funzione dedicata
```

- `X` e `Y` sono due vettori della stessa lunghezza

Nel file di script `my_dot.m`:

- Definite due vettore `X` e `Y` da usare come esempio
- Calcolatene il prodotto scalare

Confrontate il vostro risultato con quello di Matlab

- Potere sempre usare `rand`/`randi` per ottenere dei vettori di test

Laboratorio di Informatica T

Esercizio: Norma 2/Distanza Euclidea

Esercizio: Norma 2/Distanza Euclidea

Matlab fornisce una funzione per calcolare la norma di un vettore:

```
norm(X, P) % usate help per i dettagli
```

- `X` è il vettore, `P` è l'ordine della norma

Per `P = 2`, la funzione calcola la distanza euclidea dall'origine:

$$\|x\| = \sqrt{\sum_i x_i^2}$$

Nel file di script `my_norm.m`:

- Definite un vettore `X`, da utilizzare come esempio
- Calcolatene la norma

Confrontate il vostro risultato con quello di Matlab

Laboratorio di Informatica T

Esercizio: Matrice Diagonale

Esercizio: Matrice Diagonale

Matlab permette di costruire una matrice diagonale con:

```
diag(X)
```

- Restituisce una matrice diagonale
- Gli elementi sulla diagonale sono quelli del vettore **X**

$$\begin{pmatrix} x_1 & 0 & \dots & 0 \\ 0 & x_2 & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & x_n \end{pmatrix}$$

Esercizio: Matrice Diagonale

Nel file di script `my_diag.m`:

- Definite un vettore `x`, da utilizzare come esempio
- Costruite una matrice diagonale...
- ...Che abbia `x` come diagonale principale

Confrontate il vostro risultato con quello di Matlab

- Potere sempre usare `rand`/`randi` per ottenere dei vettori di test

Laboratorio di Informatica T

Esercizio: Identificazione
di Numeri Primi

Esercizio: Identificazione di Numeri Primi

Matlab fornisce la funzione:

```
isprime(N)
```

- che individua se il numero **`N`** è primo

Nel file di script **`my_isprime.m`**:

- Definite un numero intero **`X`** da usare come esempio
- Determinate se il numero sia primo, mediante l'algoritmo seguente:

$$
\begin{aligned}
&\text{prime} = \text{false} \\
&\textbf{for } d = 2..\sqrt{X} \\
&\quad \textbf{if } x \mod d = 0 \\
&\qquad \text{prime} = \text{true} \\
&\qquad \textbf{break}
\end{aligned}
$$

Esercizio: Identificazione di Numeri Primi

In matematica, l'operatore modulo:

$$z = x \mod y$$

- Denota il resto della divisione intera di x per y
- **x** è divisibile per **y** se e solo se il resto è **0**
- La divisione intera è quella che abbiamo imparato alle elementari!

Matlab permette di calcolare la divisione intera con:

```
idivide(x, y)   % Restituisce il quoziente
mod(x, y)       % Restituisce il resto
```

Esempio:

```
idivide(5, 2) % denota 2
mod(5, 2) % denota 1
```