Sopravvivere in MATLAB

R. Bernardini

14 gennaio 2003

Capitolo 1

Introduzione (breve)

1.1 Tutorial MATLAB: perché, per chi, "per come"

1.1.1 Perché (motivazione)

1.1.2 Per chi (prerequisiti)

In queste dispense verrà dato per scontato che lo studente abbia una pur minima esperienza di programmazione (questo non è un testo introduttivo alla programmazione) e le conoscenze di base di algebra lineare.

1.1.3 "Per come" (come usarle)

La struttura di queste dispense è la seguente: il Capitolo 2 introduce MATLAB "per esempi," ossia mostrando come si risolvono alcuni tipici problemi ingegneristici con MATLAB. Il Capitolo 2 non tratta in maniera approfondita la sintassi, lasciando questo compito ai capitoli che seguono.

Abbiamo scelto questo approccio poiché l'esperienza personale ci ha mostrato che è il modo più veloce (e più piacevole) per "sgrezzarsi" su un liguaggio di programmazione. A tal fine è consigliabile che lo studente esegua personalmente gli esempi presentati nei capitoli che seguono, anche quelli apparentemente più banali perché anche la semplice "battitura" degli esempi sulla tastiera aiuta, quanto meno da un punto di vista mnemonico (almeno questa è l'esperienza dell'autore).

Dove mi procuro MATLAB?

Per fare gli esercizi e gli esempi proposti è chiaramente necessario avere MATLAB. Nonostante MATLAB sia ormai diffuso ovunque in ambito universitario, lo studente potrebbe voler lavorare sul suo PC di casa, la qual cosa gli richiede di procurarsi una copia di MATLAB. Il programma, in versione "normale" è abbastanza costoso, ma, almeno fino a qualche tempo fa, dovrebbe essere disponibile una versione "per studenti," con alcune limitazioni (es. matrici di dimensione minore di 64K), ma di

costo molto minore. Maggiori informazioni possono essere trovate sul sito della Mathworks `http://www.mathworks.com`.

In alternativa, esiste un "clone" gratuito di MATLAB (Octave) che è sufficientemente compatibile con MATLAB da permettere l'esecuzione degli esempi presenti in questa dispensa. Maggiori informazioni su `http://www.octave.org/`.

Capitolo 2

Matlab per esempi

Questo capitolo introduce MATLAB per esempi. MATLAB è molto intuitivo e basta poco per "entrare in sintonia." Questo capitolo contiene diversi esempi, dal più semplice al più complesso che introducono i vari aspetti di Matlab. Idealmente, già questo capitolo dovrebbe sgrezzare abbastanza il lettore. I successivi introducono più dettagli e mostrano il linguaggio più precisamente. Il lettore si senta libero di saltare ai capitoli successivi non appena si sentirà a suo agio con gli esempi di questo capitolo.

2.1 Help!

Prima di addentrarci all'interno dei misteri di MATLAB è bene dotarsi di alcuni "dispositivi di sicurezza"

2.1.1 Come chiedere aiuto

I due principali comandi di help in MATLAB sono `help` e `lookfor`. Octave ha una struttura leggermente diversa dell'help e non ha il comando `lookfor`. Qui tratteremo solo di MATLAB.

Comando `help`

L'uso più comune del comando `help` è `help <funzione>` e mostra la sintassi di `funzione`. Esempio

```
>>> help exp
```

```
 EXP    Exponential.
    EXP(X) is the exponential of the elements of X, e to the X.
    For complex Z=X+i*Y, EXP(Z) = EXP(X)*(COS(Y)+i*SIN(Y)).
```

```
    See also LOG, LOG10, EXPM, EXPINT.
```

Dando `help` senza altri parametri si ottiene l'indice dei vari "capitoli" dell'help di MATLAB

```
>>> help

matlab/general       -  General purpose commands.
matlab/ops           -  Operators and special characters.
matlab/lang          -  Programming language constructs.
matlab/elmat         -  Elementary matrices and matrix manipulation.
matlab/elfun         -  Elementary math functions.
matlab/specfun       -  Specialized math functions.
matlab/matfun        -  Matrix functions - numerical linear algebra.
matlab/datafun       -  Data analysis and Fourier transforms.
matlab/polyfun       -  Interpolation and polynomials.
matlab/funfun        -  Function functions and ODE solvers.
matlab/sparfun       -  Sparse matrices.
matlab/graph2d       -  Two dimensional graphs.
matlab/graph3d       -  Three dimensional graphs.
matlab/specgraph     -  Specialized graphs.
matlab/graphics      -  Handle Graphics.
...
```

Con `help <nome_capitolo>` si ottiene la lista delle funzioni descritte sotto il capitolo in questione

```
>>> help elmat

  Elementary matrices and matrix manipulation.

  Elementary matrices.
    zeros       - Zeros array.
    ones        - Ones array.
    eye         - Identity matrix.
    repmat      - Replicate and tile array.
    rand        - Uniformly distributed random numbers.
    randn       - Normally distributed random numbers.
    linspace    - Linearly spaced vector.
    logspace    - Logarithmically spaced vector.
    meshgrid    - X and Y arrays for 3-D plots.
    :           - Regularly spaced vector and index into matrix.

  Basic array information.
```

```
    size        - Size of matrix.
    length      - Length of vector.
...
```

Esempio Supponiamo di voler cercare la funzione che calcola il determinante di una matrice. Non sapendo da dove iniziare, cerchiamo (con `help`) tra i "capitoli" di matlab. Troviamo "matfun - Matrix functions" che potrebbe essere quello che parla della funzione che calcola il determinante. Proviamo a vedere

```
>>> help matfun

  Matrix functions - numerical linear algebra.

  Matrix analysis.
    norm        - Matrix or vector norm.
    normest     - Estimate the matrix 2-norm.
    rank        - Matrix rank.
--> det         - Determinant.
    trace       - Sum of diagonal elements.
    null        - Null space.
    orth        - Orthogonalization.

  ...
```

Vediamo quindi che la funzione che calcola il determinante è `det`[1] e per vedere come si usa chiediamo

```
>>> help det

 DET    Determinant.
    DET(X) is the determinant of the square matrix X.

    Use COND instead of DET to test for matrix singularity.

    See also COND.
```

Comando `lookfor`

Il comando `help` è utile per vedere come si usa una funzione di cui conosciamo il nome, ma come si fa quando non sappiamo nemmeno il nome della funzione? Una possibile alternativa al cercare nell'indice di MATLAB è il comando `lookfor <stringa>` che cerca `<stringa>` nella prima riga dell'help delle funzioni disponibili.

[1] Non che ci volesse una gran fantasia ad immaginarlo...

Esempio Supponiamo di voler cercare la funzione che calcola gli autovalori e gli autovettori di una matrice. Sapendo che il termine inglese per "autovalore" è *eigenvalue*, cerchiamo con `lookfor` la stringa `eigen`.

```
>>> lookfor eigen

ROSSER Classic symmetric eigenvalue test problem.
WILKINSON Wilkinson's eigenvalue test matrix.
BALANCE Diagonal scaling to improve eigenvalue accuracy.
CONDEIG Condition number with respect to eigenvalues.
EIG    Eigenvalues and eigenvectors.
...
```

La funzione che cerchiamo si chiama `eig` e per sapere come si usa digitiamo

```
>>> help eig

    E = EIG(X) is a vector containing the eigenvalues of a square
    matrix X.

    [V,D] = EIG(X) produces a diagonal matrix D of eigenvalues and a
    full matrix V whose columns are the corresponding eigenvectors so
    that X*V = V*D.

    ...
```

2.2 Hello world

L'esempio più "classico" di programma, il solito *Hello world*, purtroppo in Matlab non è molto illuminante... Per MATLAB il corrispondente di *Hello world* potrebbe essere la risoluzione di un sistema lineare. Supponiamo di voler trovare x_1, x_2 e x_3 tali che

$$\begin{cases} x_1 + x_2 + x_3 &= 2 \\ 2x_1 + 3x_2 - x_3 &= 4 \\ x_1 - x_2 - x_3 &= 1 \end{cases} \tag{2.1}$$

Scriviamo (2.1) in forma matriciale

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{2.2}$$

dove

$$\mathbf{A} = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 3 & -3 \\ 1 & -1 & -1 \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} 2 \\ 4 \\ 1 \end{bmatrix} \tag{2.3}$$

Il nostro scopo è di risolvere (2.2), ossia calcolare $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$, in MATLAB. Prima di tutto, dobbiamo creare la matrice $\mathbf{A}$ ed il vettore $\mathbf{b}$: a tale scopo al prompt digitiamo

```
>>> b=[2; 4; 1];

>>> A = [ 1 1 1 ; 2 3 -3; 1 -1 -1]
A =

   1   1   1
   2   3  -3
   1  -1  -1
```

Ora possiamo calcolare **x** con

```
>>> x=inv(A)*b
x =

  1.500000
  0.416667
  0.083333
```

Già queste poche righe ci illustrano alcune particolarità di MATLAB: in MATLAB le matrici si scrivono tra parentesi quadre `[...]` con i punti-e-virgola '`;`' che separano le righe della matrice e gli spazi che separano elementi adiacenti sulla stesa riga. Alternativamente, gli elementi appartenenti alla stessa riga possono essere separati da virgole; per esempio, la matrice **A** poteva essere inizializzata anche nel seguente modo

```
>>> A = [ 1, 1, 1 ; 2, 3, -3; 1, -1, -1]
A =

   1   1   1
   2   3  -3
   1  -1  -1
```

o mescolando arbitrariamente virgole e spazi come in

```
>>> A = [ 1, 1 1 ; 2, 3 -3; 1 -1, -1];
```

Un'altra particolarità di MATLAB che risulta da questi primi comandi è l'effetto del '`;`' a fine comando: osservando l'inizializzazione di **A** e **b** si vede come il valore di **A**, diversamente dal valore di **b**, sia stato stampato sullo schermo. Questo perché il comando che inizializza **b** termina col '`;`', mentre il comando che inizializza **A** non ha nessun particolare terminatore. Si osservi inoltre che il valore di **x** viene stampato sullo schermo poiché non abbiamo terminato il comando con '`;`'.

Remark 1. *Un errore molto comune, talvolta abbastanza fastidioso, è dimenticarsi il '`;`' alla fine di un comando che crea una matrice molto grande. Per esempio, digitando*

```
>>> I = eye(100)
```

MATLAB *stampa sullo schermo le 10.000 componenti di una matrice identità* 100×100.

Un'ultima cosa che si deduce dal comando `x=inv(A)*b` è che l'inversa della matrice **A** si calcola con `inv(A)` e che `*` esegue il prodotto matrice-vettore.

Domanda 1. *Sommare i due vettori* $[1,3,7]$ *e* $[5,9,-1]$ *in* MATLAB*? (Non è stato detto esplicitamente come si sommano i vettori, ma è facile immaginarlo...)*

2.3 Quantizzazione

2.3.1 Introduzione

Il primo problema che si incontra nell'elaborazione numerica del segnale è dato dal fatto che mentre i segnali da elaborare sono analogici (possono cioè assumere un'infinità non numerabile di valori), il microprocessore "conosce" solo numeri interi (per giunta in quantità finita). È quindi necessario trasformare il segnale di ingresso in una sequenza di numeri interi.

Supponiamo, per esempio, di voler controllare la temperatura di un forno tramite un sistema a microprocessore. Nel sistema sarà quindi presente un dispositivo elettronico che trasforma il valore di temperatura in un valore di tensione (o di corrente, ma per fissare le idee supponiamo di lavorare con le tensioni). Perché il valore di tensione possa venir letto dal microprocessore deve essere prima trasformato in un numero intero. Il modo più semplice è di scegliere un intervallo di temperatura minimo Δ_T (supponiamo, per esempio, $\Delta_T = 0,1$ K) e di approssimare la temperatura di ingresso col più vicino multiplo intero di Δ_T. Per esempio, se la temperatura è $T = 350,231$ K il suo valore approssimato sarà $350,2 = 3502\Delta_T$. A questo punto possiamo passare al microprocessore il numero intero 3502 che rappresenta la temperatura approssimata di $350,2$ K.

Più in generale, il *quantizzatore uniforme con passo* Δ è la funzione $q_\Delta(x)$ che associa ad ogni $x \in \mathbb{R}$ il multiplo intero di Δ più vicino ad x.

2.3.2 Il problema

Supponiamo di avere un vettore **x** di misure

$$\mathbf{x} = [1.23, 2.33, 5.11, -3.67] \tag{2.4}$$

e di voler quantizzare dette misure con passo $\Delta = 0,2$. Concentriamoci, per cominciare, sul primo valore $x_1 = 1,23$. Quanto vale $q_{0,2}(1,23)$? Dato che $1,23 = 6 \cdot 0,2 + 0,03$ è facile convincersi che il multiplo di $0,2$ più vicino a $1,23$ è $6 \cdot 0,2 = 1,2$. Si osservi che 6 è l'intero più vicino a $1,23/0,2 = 6,15$.

Dall'esempio appena presentato si deduce che $q_\Delta(x)$ può essere calcolato col seguente algoritmo

1. Calcola $y = x/\Delta$

2. Sia $n = \text{round}(y)$ l'intero più vicino a y
3. Poni $q_\Delta(x) = y \cdot \Delta$.

Tale algoritmo ha un immediato corrispondente in MATLAB

```
>>> Delta=0.2;
>>> x=[ 1.23, 2.33, 5.11, -3.67 ];
>>> y=x / Delta
y =

    6.1500   11.6500   25.5500  -18.3500

>>> n=round(y)
n =

    6   12   26  -18

>>> q=n*Delta
q =

   1.2000   2.4000   5.2000  -3.6000
```

2.3.3 Commenti

I primi due comandi non dovrebbero aver bisogno di spiegazioni, gli altri tre non sono stati terminati dal ';' per mostrare l'effetto delle varie operazioni. Si osservi che dal terzo e dal quinto comando si deduce che moltiplicare o dividere un vettore (`x` o `n`) per uno scalare (`Delta`) ha l'effetto di moltiplicare o dividere ogni componente del vettore per lo scalare. Tutto ciò non dovrebbe essere troppo sorprendente.

Inoltre, dal quarto comando si deduce che passando un vettore alla funzione `round` (che calcola l'intero più vicino) si ottiene l'effetto di applicare la funzione `round` ad ogni componente. Anche questo comportamente è decisamente ragionevole.

Val la pena di osservare che tutte le funzioni "normali" (seno, coseno, logaritmo, esponenziale, ecc...) hanno questo comportamento, per esempio

```
>>> sin(x)
ans =

   0.94249   0.72538  -0.92199   0.50416

>>> log(abs(x))
ans =
```

```
  0.20701  0.84587  1.63120  1.30019
```

Infine, val la pena di osservare che nonostante l'operazione di quantizzazione sia stata più sopra "spezzata," per motivi didattici, nei suoi tre passi fondamentali, è più semplice scriverla in un solo comando, ossia

```
>>> q=Delta*round(x/Delta)
q =

   1.2000   2.4000   5.2000  -3.6000
```

2.4 Stima ai minimi quadrati

2.4.1 Il problema

Si supponga di voler misurare la resistenza elettrica di un certo conduttore. A tale scopo, abbiamo applicato alcune differenze di potenziale (1 V, 1,5 V e 3 V) ai capi del conduttore ed abbiamo misurato la corrente che attraversava il conduttore. Siano 0,6 A, 0,8 A e 1,8 A le correnti misurate. Come ben noto, per la legge di Ohm $V = RI$ le coppie tensione/corrente devono stare su una stessa retta la cui pendenza dipende dalla resistenza.

Per prima cosa verifichiamo se le tre coppie tensione/corrente stanno sulla stessa retta e a tale scopo scriviamo i valori di tensione e di corrente in due vettori `V` e `I`

```
>>> I=[0.6 0.85 1.8]
I =

  0.60000  0.85000  1.80000

>>> V=[1 1.5 3]
V =

  1.0000  1.5000  3.0000
```

tracciamo il grafico corrispondente

```
>>> plot(V,I)
```

Come si vede dal grafico i tre valori non sono allineati a causa degli errori di misura. Questo vuol dire che se calcoliamo i tre rapporti tensione/corrente dovremmo ottenere tre valori diversi. Infatti,

```
>>> V(1)/I(1)
ans = 1.6667
```

```
>>> V(2)/I(2)
ans = 1.7647
>>> V(3)/I(3)
ans = 1.6667
```

o, più brevemente (attenzione al '`.`' prima del '`/`'!)

```
>>> V ./ I
ans =

  1.6667  1.7647  1.6667
```

Onde minimizzare l'effetto degli errori è conveniente stimare il valore della resistenza ai minimi quadrati, ossia, scegliere il valore R per il quale la somma dei quadrati degli errori

$$\sum_{n=1}^{3} (V_n - RI_n)^2 \tag{2.5}$$

è minima. Se i valori V_n e I_n vengono organizzati in due vettori riga $\mathbf{v} = [V_1, V_2, V_3]$, $\mathbf{i} = [I_1, I_2, I_3]$, è ben noto[2] che la soluzione a (2.5) si scrive

$$R = \frac{\mathbf{v}\mathbf{i}^t}{\mathbf{i}\mathbf{i}^t} \tag{2.6}$$

Il calcolo di (2.6) è immediato in MATLAB

```
>>> R=(V*I')/(I*I')
R = 1.6831
```

Per controllo, disegnamo la retta relativa al valore stimato di R e confrontiamola coi dati sperimentali segnando questi ultimi con una crocetta

```
>>> plot(I, V, 'x', I, I*R)
```

2.4.2 Commenti

Questo secondo esempio ci introduce alcuni nuovi aspetti di MATLAB.

- Possiamo produrre grafici col comando `plot`. Si osservi nell'ultimo uso di `plot` come l'uso di `'x'` dopo la coppia `I, V`, ci permetta di marcare i valori sperimentali con crocette. `plot` è in realtà un comando molto versatile e si invita il lettore a consultare l'help in linea con `help plot`.
- L'operatore '`./`' (un punto subito seguito dal segno di divisione) esegue la divisione "componente-a-componente" di due vettori (o matrici). La presenza del "punto" prima dell'operatore serve a chiedere a MATLAB di eseguire l'operazione componente per componente (o "punto a punto") e non in modo "vettoriale." Si provi, per esempio, a vedere la differenza tra `A .* A` e `A * A` (con `A` uguale alla matrice dell'esempio in Sezione 2.2).

[2] Vedi Appendice

- L'apostrofo `''` è l'operatore di trasposizione.

```
>>> V
V =

  1.0000  1.5000  3.0000

>>> V'
ans =

  1.0000
  1.5000
  3.0000
```

Domanda 2. *Cosa avremmo ottenuto digitando* `V/I` *senza il punto? Che senso ha una divisione tra vettori? Si provi ad eseguire* `V/I` *e si cerchi di interpretare il risultato. Lo abbiamo già visto da qualche parte? (Per maggiori dettagli si consulti l'help in linea con '*`help /`*'.)*

2.5 Minimi quadrati non lineari

Per questo esempio ci serve sapere che la corrente I che passa attraverso un diodo che ha ai suoi capi una differenza di potenziale V è pari a

$$I = J_0 \left(\exp\left(\frac{V}{V_T} \right) - 1 \right) \tag{2.7}$$

dove $V_T \approx 8.6 \cdot 10^{-5} T$, dove T è la temperatura in gradi Kelvin.

Supponiamo di aver misurato la caratteristica tensione/corrente di un diodo in 5 punti e di non sapere né la temperatura a cui sono state fatte le misure, nè J_0. Vogliamo stimare ai minimi quadrati questi due parametri, ossia vogliamo trovare J_0 e T tali che la somma dei quadrati degli errori

$$\sum_{n=1}^{5} (i_n - J_0 \exp(v_n / V_T))^2 \tag{2.8}$$

sia minima.

Per nostra comodità organizziamo le 5 misure in due vettori

$$\mathbf{v} = \begin{bmatrix} 0.600 \\ 0.650 \\ 0.700 \\ 0.750 \\ 0.800 \\ 0.850 \end{bmatrix} \qquad \mathbf{i} = \begin{bmatrix} 0.021 \\ 0.018 \\ 0.046 \\ 0.154 \\ 0.504 \\ 1.714 \end{bmatrix} \tag{2.9}$$

Per capire come risolvere questo problema, supponiamo per il momento di conoscere T. Se questo fosse vero, potremmo calcolare V_T ed i valori intermedi

$$u_i = \exp\left(\frac{v_i}{V_T}\right) - 1 \tag{2.10}$$

ed il problema si ridurrebbe a trovare il valore di J_0 che minimizza

$$\sum_{n=1}^{6} (i_n - J_0 u_n)^2. \tag{2.11}$$

Il problema è formalmente identico al problema (2.5) affrontato in Sezione 2.4 (in questo caso J_0 gioca il ruolo che giocava R in (2.5)) e la soluzione è[3]

$$J_0 = \frac{\mathbf{i}^t\mathbf{u}}{\mathbf{u}^t\mathbf{u}} \tag{2.12}$$

Purtroppo la (2.12) non è applicabile poiché non conosciamo T e quindi non possiamo calcolare i valori u_n. Possiamo però andare per tentativi e provare un certo insieme di valori (potremmo chiamarle "ipotesi") di T, per ognuno di questi valori calcolare la J_0 ottima tramite la (2.12) e scegliere la T alla quale corrisponde il minimo errore.

Riassumendo, possiamo risolvere il problema di minimo in questo modo:

1. Per ogni T in un certo intervallo (per esempio da 400 K a 500 K con incrementi di mezzo grado)
   (a) calcola i corrispondenti valori u_n tramite la (2.10)
   (b) calcola il valore ottimo di J_0 tramite la (2.12)
   (c) calcola l'errore corrispondente e salvalo in un vettore
2. Trova l'errore minimo e scegli la T corrispondente

Il codice MATLAB per implementare quest'algoritmo è abbastanza semplice

```
v=[0.600; 0.650;  0.700;  0.750;  0.800;  0.850];   % Tensioni
i=[0.021; 0.018;  0.046;  0.154;  0.504;  1.714];   % Correnti

err=[];                  % Conterrà gli errori

for T=400:500
  VT=8.6e-5*T;           % Calcola VT
  u=exp(v/VT)-1;         % Calcola i valori intermedi ui
  J0=(i'*u)/(u'*u);      % Trova la J0 ottima
```

[3] La forma di (2.12) è leggermente diversa da quella di (2.6) poichè in (2.6) i vettori sono vettori riga, mentre in (2.12) sono vettori colonna.

```
  err=[err norm(i-J0*u)]; % Salva l'errore nel vettore 'err'
end

plot(400:500, err);       % Disegna il grafico temperatura/errore
```

Si vede immediatamente dal grafico che il minimo si raggiunge per $T = 478$ K a cui corrisponde $J_0 = 1.79 \cdot 10^{-9}$ A (i valori "veri" erano $T = 465$ K e $J_0 = 10^{-9}$ A, la discrepanza è dovuta al rumore aggiunto ai dati).

2.5.1 Commenti

Questo esempio introduce un certo numero di novità. La prima è il carattere `%` che in MATLAB introduce il commento. La seconda novità è rappresentata dalla linea

```
err=[];
```

che inizializza il vettore `err` (che conterrà i valori degli errori) col vettore vuoto.

Val la pena anche di segnalare l'uso della funzione `norm` che calcola la norma (euclidea) del vettore parametro, ossia, se `x` è un vettore colonna, `norm(x)` è pari a `sqrt(x'*x)`. Si osservi quindi che il vettore `err` non contiene in realtà la somma dei quadrati degli errori, ma la radice quadrata di detta somma. In realtà questo non è un problema, dato che siamo interessati a trovare l'errore minimo e la radice quadrata è una funzione monotona.

La novità però forse più interessante è data dalle linee

```
for T=400:500
 ...
end
```

che rappresentano la prima struttura di controllo di MATLAB che incontriamo: il ciclo `for`. È abbastanza intuitivo che il "corpo" del ciclo dell'esempio viene eseguito 101 volte con `T` che assume i valori `400`, `401`, `402`, e così via fino a `500`. Se avessimo voluto usare incrementi di mezzo grado, avremmo scritto il "passo" 0.5 K tra i due estremi, ossia

```
for T=400:0.5:500
 ...
end
```

In questo caso i valori assunti da `T` sarebbero stati `400`, `400.5`, `401`, ..., `500`.

Fin qua il ciclo `for` del MATLAB sembra essere l'esatto corrispondente del ciclo `FOR` del BASIC o del ciclo `DO` del FORTRAN, solo con una sintassi un po' diversa. In realtà il ciclo `for` in MATLAB ha una particolarità che, nonostante non venga sfruttata di frequente, è abbastanza interessante.

Per capire di che si tratta, provate ad eseguire il codice che segue e cercate di capire la "regola" sottostante prima di proseguire con la lettura.

```
for x=[1, 3, 5, 7]
  x
end

for c='ciao'
  c
end

for x=[2; 4; 1]
  x
end

A = [ 1 1 1 ; 2 3 -3; 1 -1 -1];
for a=A,  a,  end
```

Incidentalmente, si osservi nell'ultimo esempio l'uso della virgola '`,`' per separare più comandi sulla stessa linea, dato che se avessimo usato il punto e virgola '`;`' il MATLAB non avrebbe stampato il valore di `a` ad ogni iterazione.

Individuata la regola? Verificatela prevedendo l'uscita del comando che segue.

```
for c=[ 'ciao '; 'pippo' ], c, end
```

La regola è la seguente: dopo il segno di '`=`' il MATLAB si aspetta una *matrice* ed esegue *un'iterazione per ogni colonna* della matrice stessa. Ecco perché il ciclo `for x=b` esegue una sola iterazion e non tre, come verrebbe fatto di aspettarsi. Si osservi inoltre che il ciclo sulla stringa `for c='ciao'` funziona perché in MATLAB le stringhe sono vettori riga di caratteri.

Quest'osservazione ci suggerisce che `400:500` nel ciclo `for` dell'esempio non faccia parte della sintassi del `for`, ma che sia un modo per creare un vettore riga. Verifichiamolo (con un vettore più piccolo)

```
>>> 400:410
ans =

  400  401  402  403  404  405  406  407  408  409  410
```

verifichiamo anche l'effetto del "passo"

```
>>> 400:0.5:403
ans =

  400.00  400.50  401.00  401.50  402.00  402.50  403.00
```

Grazie a quest'ultima osservazione possiamo riscrivere il codice dell'esempio in un modo leggermente diverso, ossia

```
v=[0.600; 0.650;  0.700;  0.750;  0.800;  0.850];
i=[0.021; 0.018;  0.046;  0.154;  0.504;  1.714];

err=[];

temperature=400:500;      % <--- NUOVA LINEA

for T=temperature         % <--- CAMBIATA
  VT=8.6e-5*T;
  u=exp(v/VT)-1;
  J0=(i'*u)/(u'*u);
  err=[err norm(i-J0*u)];
end

plot(temperature, err);   % <--- CAMBIATA
```

Si osservi come ora le temperature da provare siano contenute in vettore `temperature` e come sia possibile, grazie al comportamento del `for` in MATLAB, scrivere `for T=temperature` invece di `for T=400:500`

Questo modo di scrivere il codice ha diversi vantaggi. Un primo (minore) vantaggio è che la leggibilità è leggermente migliorata, essendo chiaro che il ciclo "gira" su un insieme di temperature. Un secondo (piccolo) vantaggio è che è un po' più facile cambiare l'insieme delle temperature provate. Si supponga, per esempio, di voler cambiare il passo da 1 K a 0,5 K. Nella prima versione si sarebbe dovuto cambiare sia la linea col `for` che quella col `plot` per mantenere le due istruzioni "sincronizzate." Nella seconda versione, invece, è sufficiente cambiare `temperature=400:500` con `temperature=400:0.5:500` perché il ciclo `for` ed il `plot` restano comunque "sincronizzati."

Un terzo vantaggio (decisamente più interessante) della seconda versione è la possibilità di far determinare a MATLAB la temperatura ottima, invece di doverla determinare noi per ispezione visiva del grafico. È sufficiente fare

```
>>> [val, idx]=min(err)
val = 0.018538

idx = 79

>>> T0=temperature(idx)
T0 = 478
```

Si osservi come il primo comando trovi l'errore minimo (`val`) e la sua posizione (`idx`) all'interno del vettore `err`. La posizione `idx` dell'errore minimo in `err` viene usata per estrarre da `temperature` la temperatura `T0` corrispondente.

2.5.2 Funzioni che restituiscono più valori

Si osservi inoltre come nell'ultimo esempio la funzione `min` restituisca due valori: il minimo trovato (`val`) e la sua posizione (`idx`). In effetti, le funzioni in MATLAB, diversamente dalla maggior parte degli altri linguaggi, possono restituire più valori che possono essere assegnati a diverse variabili usando la sintassi

```
[var1, var2, ..., varN] = funzione(...)
```

dove `funzione` è una funzione che restituisce `N` valori che vengono assegnati alle `N` variabili `var1`...`varN`.

Un altro esempio di funzione che restituisce più di un valore è la funzione `eig` che restituisce autovalori ed autovettori di una matrice. Digitando

```
>>> A = [ 1 1 1 ; 2 3 -3; 1 -1 -1];
>>> [autovec, autoval] = eig(A)
autovec =

  -0.86373  -0.39078  -0.27217
   0.20074   0.56048  -0.95258
  -0.46225   0.73018   0.13608

autoval =

   1.30278   0.00000   0.00000
   0.00000  -2.30278   0.00000
   0.00000   0.00000   4.00000
```

otteniamo in `autovec` una matrice le cui colonne sono gli autovettori di `A`, mentre in `autoval` otteniamo una matrice diagonale con gli autovalori di `A` sulla diagonale principale.

Val la pena di osservare che se una funzione restituisce più valori non è necessario assegnarli tutti. Per esempio, se siamo interessati solo al valore minimo di `err`, ma non ci interessa la sua posizione, possiamo usare

```
>>> val = min(err)
val = 0.018538
```

e il valore della posizione viene perso... Similmente, se di una matrice ci interessano solo gli autovalori, possiamo usare

```
>>> autoval= eig(A)
autoval =

   1.3028
  -2.3028
   4.0000
```

Si noti che il comportamento di una funzione MATLAB può cambiare a seconda del numero di parametri di uscita che vengono assegnati. Per esempio, `eig` restituisce gli autovalori sulla diagonale principale di una matrice quando due parametri di uscita vengono assegnati, ma li mette in vettore colonna quando un solo parametro di uscita viene assegnato.

In generale non c'è una regola fissa per prevedere il comportamento della funzione a seconda del numero di parametri di uscita e conviene fare riferimento all'help in linea della funzione.

2.5.3 Riassunto

Questa sezione è stata abbastanza "densa" di novità e conviene fare un breve riassunto dei concetti visti. I concetti più importanti sono segnati da un '⇒'.

- Il carattere '`%`' per i commenti
- La matrice vuota `[]`
- Le funzioni `norm`, `min`, `eig`
- Il fatto che in MATLAB le stringhe sono vettori riga di caratteri.
- ⇒ Il ciclo `for` (ed il fatto che "giri" sulle colonne)
- ⇒ La creazione di un vettore con il '`:`' (es. `400:500` o `400:0.5:500`)
- ⇒ Il fatto che le funzioni MATLAB possono restituire più valori

Domanda 3. *In molte applicazioni ha interesse sapere se la successione di potenze* $\mathbf{A}^n$, $n = 0, 1, \ldots$ *converge a zero o no. Si dimostra che detta successione converge a zero se e solo se l'autovalore dominante di* **A** *(ossia, quello col modulo più grande) è in modulo minore di 1. Scrivere il codice* MATLAB *(basta una riga) per calcolare il* modulo *dell'autovalore dominante di* **A**. *Leggermente più difficile: scrivere il codice* MATLAB *per calcolare* l'autovalore dominante *(ora servono tre righe).*

Capitolo 3

Nozioni base

Questo capitolo e quelli che seguono vogliono essere un breve riassunto della sintassi e delle potenzialità di Matlab.

3.1 Caratteri di "fine comando"

Ogni comando in MATLAB può essere terminato dal '`;`', dalla '`,`' o dal "fine linea". Se l'istruzione è terminata da '`,`' o dalla fine della linea, un eventuale risultato del comando verrà stampato sul terminale, se l'istruzione è terminata da '`;`' nessuna stampa avrà luogo.

Un comando può essere spezzato su più linee terminando ogni linea, salvo l'ultima, con tre punti '`...`'. Esempio:

```
>> a= 3 + sin(3)+ sqrt(44/5) + log(33) + ...
22*log(9)
a =
   57.9430
```

Più comandi possono essere scritti sulla stessa linea terminandoli con '`,`' o '`;`'. Esempio:

```
>> a=44*3, b=2*pi; c=15
a =
   132
c =
    15
```

3.2 Variabili predefinite

MATLAB ha alcune variabili predefinite

- `pi` che contiene il valore di π

- `i` e `j` che contengono l'unità immaginaria $\sqrt{-1}$

Si osservi che `pi`, `i` e `j` sono considerate delle *variabili* ed è quindi possibile cambiarne il valore. Per esempio,

```
>> pi
pi =
    3.1416
>> pi=9;
>> pi
pi =
     9


>> i*i
ans =
    -1
>> i=3;
>> i*i
ans =
     9
```

Un'altra variabile "automatica" in MATLAB è `ans` che contiene il risultato dell'ultima espressione "non assegnata", per esempio

```
>> 3+10
ans =
    13
>> ans
ans =
    13
>> ans-2
ans =
    11
>> a=4+5
a =
     9
>> ans
ans =
    11
```

Nel primo comando, il valore di `10+3` non viene esplicitamente assegnato a nessuna variabile e quindi MATLAB lo scrive in `ans` Dal terzo comando si vede come `ans` sia una variabile a tutti gli effetti e possa essere usata in espressioni. Il quarto comando assegna ad `a` il valore di `4+5` e quindi la variabile `ans` non viene toccata.

3.3 Creazione di matrici

In MATLAB è possibile creare matrici

1. Scrivendole tra [...] separando rige adiacenti col ';' e gli elementi della stessa riga con ',' o spazi. Per esempio, `[1, 2 ; 3 4]` corrisponde alla matrice

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}. \tag{3.1}$$

   La scrittura tra [...] può anche essere usata per costruire matrici " blocchi," per esempio

```
>> a=[1, 2; 3, 4]
a =
     1     2
     3     4
>> b=[5 ; 6], c=[0 9]
b =
     5
     6
c =
     0     9

>> [a b ; c 0]
ans =
     1     2  |  5
     3     4  |  6
     ---------+---
     0     9  |  0
```

   (Nota: la divisione dei blocchi nell'ultima matrice è stata aggiunta "a mano").

2. Usando l'operatore ':' per creare vettori riga "regolari"; per esempio,

```
>> c=3:10
c =
     3     4     5     6     7     8     9    10

>> d=3:2:9
d =
     3     5     7     9

>> b=3:0.3:4
```

```
b =
    3.0000    3.3000    3.6000    3.9000
```

Si noti come la forma con tre argomenti permetta di cambiare il "passo".

3. Usando funzioni specializzate, tra cui `ones`, `zeros`, `rand`, `randn` e `eye` Ognuna di queste funzioni può essere chiamata con due parametri interi (es. `ones(N,M)`), un parametro vettoriale (es. `rand([N, M])`) o un parametro intero (es. `zeros(N)`). Nel primo caso e nel secondo caso viene creata una matrice con `N` righe ed `M`, mentre nel terzo crea una matrice `N`$\times$`N` (ossia, per default, $N = M$). Le funzioni si differenziano per il tipo di matrice creato, più precisamente

   - `ones` e `zeros` creano matrici aventi tutte le componenti pari, rispettivamente, a 1 e a 0.
   - `eye(N)` crea una matrice `N`$\times$`M` avente 1 sulla diagonale principale e 0 altrove.
   - `rand` e `randn` creano matrici casuali. `rand` genera le componenti secondo una distribuzione uniforme tra 0 e 1, mentre `rand` genera le componenti secondo una distribuzione gaussiana con media zero e varianza unitaria.

   Esempio:

```
>> a=eye(3,2)
a =
     1     0
     0     1
     0     0
>> b=ones(2)
b =
     1     1
     1     1
>> c=zeros([2, 3])
c =
     0     0     0
     0     0     0
>> d=rand(size(b))
d =
    0.9501    0.6068
    0.2311    0.4860
```

   Si noti nell'esempio l'uso della funzione `size` per "leggere" le dimensioni di `b`

```
>> size(a)
ans =
     3     2
```

Un'altra funzione molto utile per la creazione di matrici è `diag` che può essere chiamata in due modi. Con `diag(d)`, dove `d` è un vettore, crea una matrice diagonale avente gli elementi di `d` sulla diagonale principale. Esempio:

```
>> diag(1:3)
ans =
     1     0     0
     0     2     0
     0     0     3
```

Se il parametro passato a `diag` è invece una matrice `M`, `diag(M)` restituisce un vettore colonna le cui componenti sono gli elementi sulla diagonale principale di `M`, per esempio

```
>> a=rand(3)
a =
    0.8913    0.0185    0.6154
    0.7621    0.8214    0.7919
    0.4565    0.4447    0.9218
>> diag(a)
ans =
    0.8913
    0.8214
    0.9218
```

3.4 Accesso alle matrici

3.4.1 Accesso ai singoli elementi

Se `v` è un vettore `v(n)` è l'`n`–sima componente di `v`, mentre se `A` è una matrice, `A(r,c)` è l'elemento di riga `r` e colonna `c`. Dalla versione 5 in poi nelle espressioni per `r`, `c` ed `n` si può usare la parola chiave `end` che viene ad assumere il valore dell'ultimo indice, per esempio, se

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

`A(2,end)` restituisce l'ultimo elemento della seconda riga

```
>> A(2,end)
ans =
     6
```

mentre `A(end-1, end-1)` restituisce il penultimo elemento della penultima riga.

```
>> A(end-1,end-1)
ans =
     5
```

Abbastanza stranamente, se `A` è una matrice, è MATLAB accetta anche `A(n)`; in questo caso `A` viene considerato in vettore costruito "impilando" le colonne di `A`, per esempio

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

Si osservi come quando MATLAB incontrando espressioni del tipo `A(n)` consideri `A` come il vettore colonna ottenuto impilando le colonne di `A`, ossia

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

3.4.2 Estrazione di sotto-matrici

In `A(r,c)` e `v(n)`, i valori `n`, `r` e `c` possono essere anche vettori di interi. In questo caso il risultato è la sottomatrice. Per esempio, se

```
>> B=magic(5)
B =
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
```

`B(1:3, 2)` estrae le prime tre righe della seconda colonna

```
>> B(1:3, 2)
ans =
    24
     5
     6
```

`B(1:2:end, 1:2:end)` estrae la sottomatrice aventi gli elementi corrispondenti a righe e colonne dispari (si noti l'uso di `end`)

```
>> B(1:2:end, 1:2:end)
ans =
    17     1    15
     4    13    22
    11    25     9
```

`B(3, 3:end-1)` estrae la penultima e la terzultima colonna della terza riga (si noti l'uso di `end-1` per indicare la penultima colonna)

```
>> B(3, 3:end-1)
ans =
    13    20
```

L'uso di '`:`' al posto di un indice equivale ad usare `1:end`. Per esempio, `B(1:3,:)` seleziona le prime tre righe della matrice `B`

```
>> B(1:3,:)
ans =
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
```

In particolare, scrivendo `A(:)` si ottiene un vettore colonna ottenuto "impilando" le colonne di `A`. Per esempio,

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

Si osservi che `A(:)` è *sempre un vettore colonna*, anche se `A` era originariamente un vettore riga. L'uso della notazione `(:)` è quindi molto utile per "forzare" un vettore ad essere un vettore colonna

```
>> v=1:3,  u=[1; 4; 9]
v =
     1     2     3
u =
     1
     4
     9
>> v(:), u(:)
ans =
     1
     2
     3
ans =
     1
     4
     9
```

Gli indici vettoriali possono essere usati anche per sovrascrivere parti di una matrice; per esempio, per sostituire le prime tre righe e tre colonne di `B` con la matrice identità possiamo usare

```
>> B(1:3, 1:3)=eye(3)
B =
     1     0     0     8    15
     0     1     0    14    16
     0     0     1    20    22
    10    12    19    21     3
    11    18    25     2     9
```

Si osservi sul lato destro del segno di '=' ci deve essere una matrice di dimensioni pari a quelle della sottomatrice selezionata a sinistra del segno '=' o uno scalare. In quest'ultimo caso tutti gli elementi della sottomatrice sono posti uguali allo scalare. Per esempio, per porre uguale a zero la quarta riga di B possiamo usare

```
>> B(4, :)=0
B =
     1     0     0     8    15
     0     1     0    14    16
     0     0     1    20    22
     0     0     0     0     0
    11    18    25     2     9
```

Se alla destra di '=' c'è la matrice vuota [] la sottomatrice a sinistra viene rimossa. Per esempio, per rimuovere la terza colonna di B possiamo usare

```
>> B(:, 3)=[]
B =
     1     0     8    15
     0     1    14    16
     0     0    20    22
     0     0     0     0
    11    18     2     9
```

Quest'ultima caratteristica è molto utile per "sottocampionare" vettori, per esempio se

```
>> v=rand(1,7)
v =
    0.4447    0.6154    0.7919    0.9218    0.7382    0.1763    0.4057
```

v decimato di un fattore due si ottiene con

```
>> v(1:2:end)
u =
    0.4447    0.7919    0.7382    0.4057
```

Se vogliamo "interpolare" u inserendo degli zeri tra una componente e l'altra, possiamo usare

```
>> clear t
>> t
??? Undefined function or variable 't'.


>> t(1:2:length(u)*2)=u
t =
    0.4447         0    0.7919         0    0.7382         0    0.4057
```

Si noti l'uso di `clear t` per "cancellare" `t` (nel caso fosse già stata presente in memoria) e di `length(u)` per determinare il numero di componenti di `u`. Si osservi inoltre che i valori di `t` non assegnati dalla seconda istruzione (quelli con componenti pari) vengono messi a zero di default. Nel caso `t` fosse esistita al momento dell'assegnazione, le componenti non assegnate avrebbero conservato il loro valore, per esempio

```
>> q=1:3
q =
     1     2     3
>> q(1:2:length(u)*2)=u
q =
    0.4447    2.0000    0.7919         0    0.7382         0    0.405
```

Si osservi come la seconda componente di `q` abbia conservato il suo valore `2` e come la quarta e la sesta (che non esistevano) siano state poste pari a `0`.