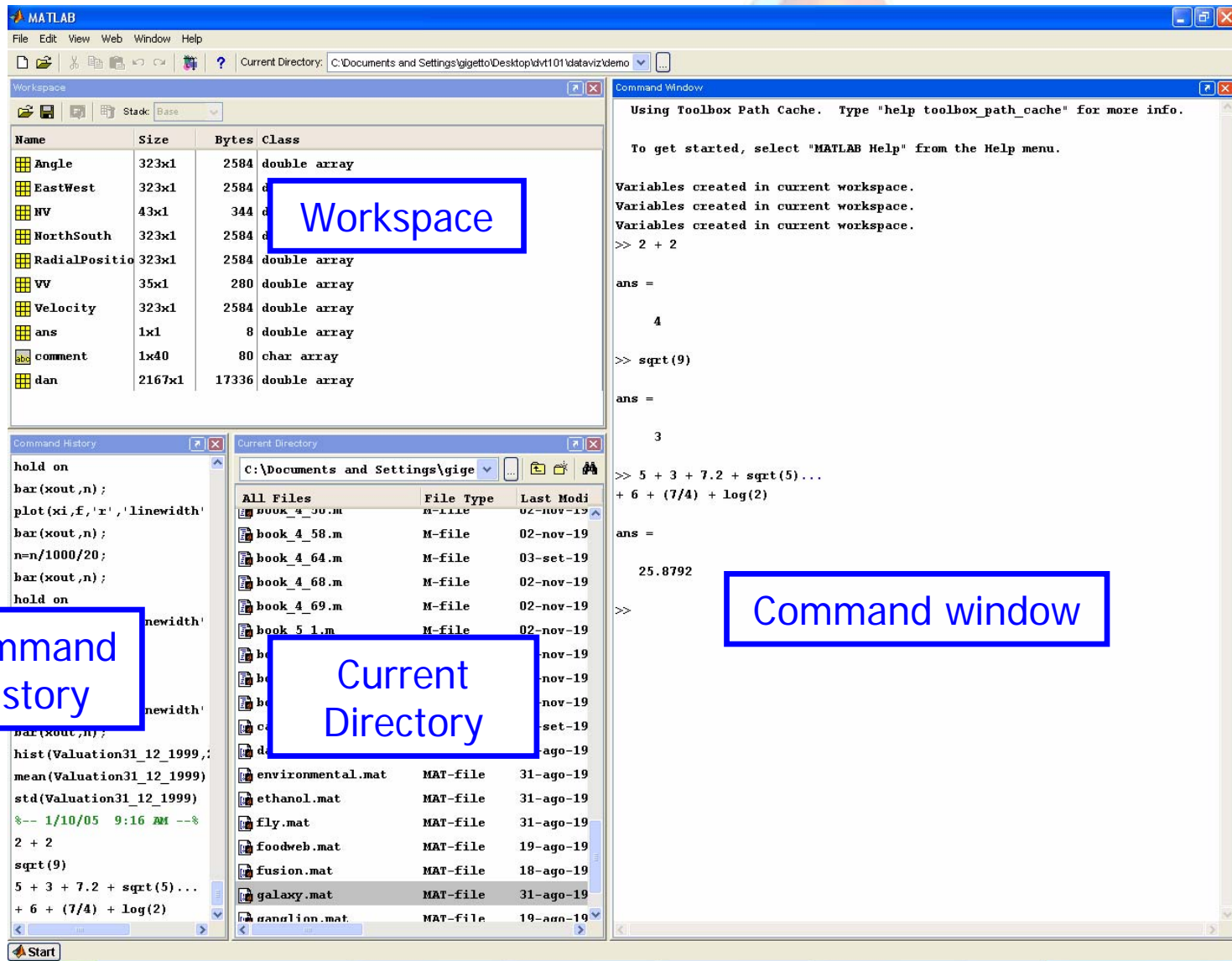


Introduction to MATLAB

Gianfranco Forte

*University Bocconi, Financial Markets and Institutions
Department.*

The four panels of the desktop



The command window as a calculator

» (3+5)/7

*Sum of (3+5)
divided by 7*

ans =

1.1429

+	<i>addition</i>
-	<i>subtraction</i>
/	<i>division</i>
*	<i>multiplication</i>
^	<i>power</i>

» x = (3+5)/7

define a variable

x =

1.1429

» x

x =

1.1429

*Type the name of the
variable to display it in the
command window*

The command window as a calculator



- ✓ When you are working in the Command Window, you can recall the last command you have run using the up arrow of the **keyboard**; typing this key another time you recall the code executed before the last one, and so on. Remember that you can also recall commands recorded in the Command History (commands run in a previous work session).

E.g

Arrays and Matrices in MATLAB (The MATrix LABoratory)



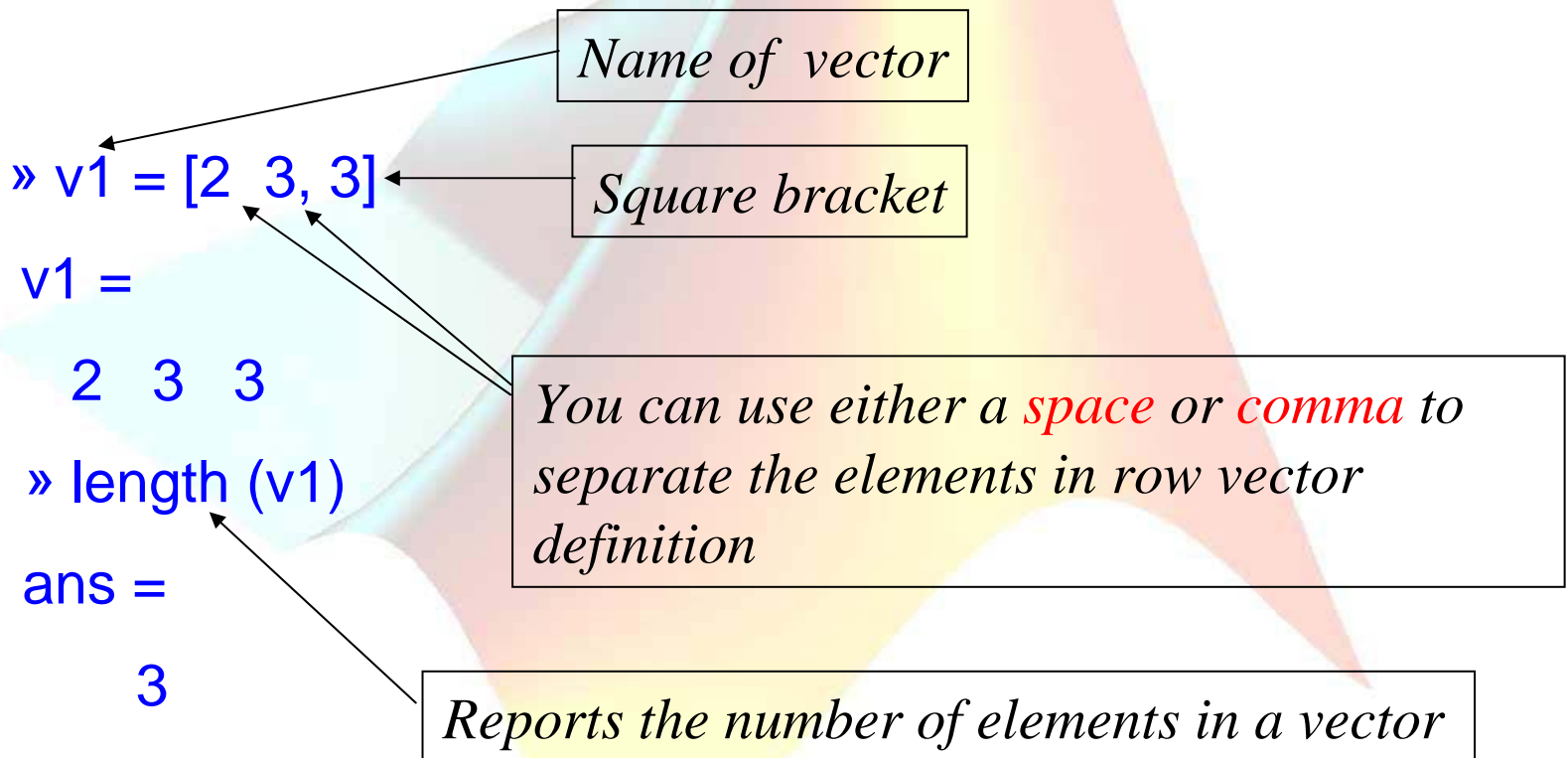
A **matrix** is a two-dimensional array with special rules for addition, multiplication, and other operations. It represents a mathematical linear transformation. The two dimensions are called the **rows** and the **columns**. A **vector** is a matrix for which one dimension has only the index 1. A **row vector** has only one row and a **column vector** has only one column.

Note: in MATLAB there is really no formal distinction — not even between a scalar and a 1×1 matrix. Try to see it in the workspace (e.g.....)

The MATrix LABoratory

The simplest way to construct a small array is by enclosing its elements in square brackets.

Defining a vector (Row vector)



Standard operations

You can perform standard linear algebra operations on these row vector

$$\gg v2 = 3*v1$$

Multiplication by a scalar quantity

$$v2 =$$

$$6 \quad 9 \quad 9$$

$$\gg v3 = [2 \quad 2]$$

$$v3 =$$

$$2 \quad 2$$

$$\gg v4 = v3+v1$$

Other vector operations:

- Addition/subtraction of a vector

-...etc

*** Row vectors must have the same length
(more on this later)*

??? Error using ==> +
Matrix dimensions must agree.

Defining a vector (Column vector)

You can also define a **column vector**

```
» c1 = [1;3;2]
```

Square bracket

```
c1 =
```

```
1  
3  
2
```

Use **semicolon** or enter the value at the **next line** for next row elements

```
» c2 = [2  
2  
4]
```

```
c2 =
```

```
2  
2  
4
```

For a matrix 3 by 3:

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

```
A =
```

```
1 2 3  
4 5 6  
7 8 9
```


Concatenation of matrices or vectors

Concatenation: to join **small** (compatible) matrices

to make bigger ones:

```
>> A = magic(4)
```

A =

16	2	3	13
5	11	10	8
9	7	6	12
4	14	15	1

```
>> B = [A A-2; A*2 A/4];
```

```
>> C = [A B'];
```

C =

16	2	3	13	4
5	11	10	8	7
9	7	6	12	5
4	14	15	1	1

The dimensions of the vectors or matrices you want to concatenate must be compatible !!!!!

```
>> B = [4 7 5 1]
```

B =

4	7	5	1
---	---	---	---

```
% Try the concatenation
```

```
>> C = [A B];
```

??? Error using ==> horzcat
All matrices on a row in the bracketed expression must have the same number of rows.

e.g. vertical concatenations
(same number of columns).....

Matrix operations

The arithmetic operators $+$, $-$, $*$, $^$ are interpreted in a matrix sense. When appropriate, scalars are “**expanded**” to match a matrix.

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

```
A =
```

```
1 2 3
4 5 6
7 8 9
```

```
>> b = [0;1;0]
```

```
b =
```

```
0
1
0
```

```
>> A+A
```

```
ans =
```

```
2 4 6
8 10 12
14 16 18
```

```
>> ans-1
```

```
ans =
```

```
1 3 5
7 9 11
13 15 17
```

```
>> 3*B
```

```
ans =
```

```
0
3
0
```

Matrix information commands.

<code>size</code>	size in each dimension
<code>Length</code>	size of longest dimension (number of elements for vectors)
<code>ndims</code>	number of dimensions
<code>find</code>	indices of nonzero elements

Matrix operations

The following matrix operations are available in MATLAB:

- Plus	+
- Minus	-
- Matrix multiply	*
- Matrix power	^
- Backslash or left matrix divide	\
- Slash or right matrix divide	/
- Kronecker tensor product	kron
- Conjugate transpose	'

```
>> b*A
??? Error using ==> *
Inner matrix dimensions must agree.
```

```
>> A = [1 2 3; 4 5 6; 7 8 9]
A =
     1     2     3
     4     5     6
     7     8     9
>> b = [0;1;0]
b =
     0
     1
     0
```

These matrix operations **apply, of course, to scalars** (1 X 1 matrices) as well. If the sizes of the matrices are incompatible for the matrix operation, an error message will result, except in the case of **scalar - matrix operations** (for addition, subtraction, and division as well as for multiplication) in which case each entry of the matrix is operated on by the scalar.

Matrix operations

(Matrix – Matrix) operations

- Plus	+	Same dimensions
- Minus	-	Same dimensions
- Matrix multiply	*	matrix dimensions must agree
- Matrix power	^	Impossible
Right matrix divide	/	matrix dimensions must agree
Left matrix divide	\	matrix dimensions must agree

matrix dimensions must agree

$$(n\text{-by-}p) * (p\text{-by-}m)$$

$(n\text{-by-}m)$

Only a **squared matrix** and a scalar

Pay attention!

Colon notation

Colon notation, which is used both to generate vectors and reference submatrices and subscripting are keys to efficient manipulation of data.

Creative use of these features to vectorize operations permits one to minimize the use of loops (which slows MATLAB) and to make code simple and readable.

Special effort should be made to become familiar with them.

The expression `>> 1:5` is actually the row vector `[1 2 3 4 5]`.

The numbers need not be integers nor the increment one. For example:

<code>>> 0.2:0.2:1.2</code>	gives	<code>[0.2, 0.4, 0.6, 0.8, 1.0, 1.2]</code> , and
<code>>> 5:-1:1</code>	gives	<code>[5 4 3 2 1]</code> .

Linspace

`linspace` Generate linearly spaced vectors

Syntax

`y = linspace(a,b)`

`y = linspace(a,b,n)`

The `linspace` function generates linearly spaced vectors. It is similar to the colon operator `:`, but gives **direct control over the number of points**.

`y = linspace(a,b)` generates a row vector `y` of 100 points linearly spaced between and including `a` and `b`.

`y = linspace(a,b,n)` generates a row vector `y` of `n` points linearly spaced between and including `a` and `b`.

Array operations

The matrix operations of **addition** and **subtraction** already operate entry-wise but the other matrix operations given above do not. They are matrix operations. It is important to observe that these other operations, *****, **^**, ****, and **/**, can be made to operate entry-wise by **preceding them by a period**. For example, either `[1,2,3,4].*[1,2,3,4]` or `[1,2,3,4]^2` will yield `[1,4,9,16]`.

- Array multiply `.*`
- Array power `.^`
- Left array divide `.\`
- Right array divide `./`

Remember that, when you work on an element by element basis, the dimensions of the matrices or vectors involved **must be the same**

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

```
A =
```

```
1 2 3
4 5 6
7 8 9
```

```
>> C = [1 3 -1; 2 4 0; 6 0 1];
```

```
C =
```

```
1 3 -1
2 4 0
6 0 1
```

```
>> A.* C
```

```
ans =
```

```
1 6 -3
8 20 0
42 0 9
```

Try for A*C.

Array operations (example)

You need to evaluate this function on the range [0,1].

$$f(x) = \frac{120 - 60x + 12x^2 - x^{-3}}{120 + 60x + 12x^2 + x^{-3}}$$

```
>> x = linspace(0,1,5); % or x = 0:0.25:1;  
>> y = (120-60*x+12*x.^2- x.^3)./(120+60*x+12*x.^2+x.^3);  
>> [x'y']
```

```
ans =  
  
0      1.0000  
0.2500 0.7788  
0.5000 0.6065  
0.7500 0.4724  
1.0000 0.3679
```

```
>> x = linspace(0,10,30);  
>> y = (120-60*x+12*x.^2-...  
x.^3)./(120+60*x+12*x.^2+x.^3);  
>> Tab = [x' y'];  
>> plot(Tab(:,1),Tab(:,2));
```

Variable on
the X- axis

Variable on
the Y- axis

E.g: Calculating returns from prices of stocks

Load the prices of italian stocks that you find in the file
`"prices.wk1"`.

Try to calculate arithmetic returns from the prices.

- (1) Do it using a `for` loop.
- (2) After that try to do the same, using the possibility to operate `element by element` on the matrix of prices.

Matrix Indexing in MATLAB

Indexing into a matrix is a means of selecting a **subset of elements from the matrix**. MATLAB has several indexing styles that are not only powerful and flexible, but also readable and expressive. Indexing is a key to MATLAB's effectiveness at capturing matrix-oriented ideas in understandable computer programs.

Indexing is also closely related to another term MATLAB users often hear: **vectorization**. Vectorization means using MATLAB language constructs to eliminate program loops, usually resulting in programs that run faster and are more readable.

Indexing Vectors (1)

Let's start with the simple case of a vector and a single subscript. The vector is

```
v = [16 5 9 4 2 11 7 14];
```

The subscript can be a single value.

```
v(3) % Extract the third element  
ans =  
     9
```

Or the subscript can itself be another vector.

```
v([1 5 6]) % Extract the first, fifth, and sixth elements  
ans =  
    16     2    11
```

Indexing Vectors (2)

```
v = [16 5 9 4 2 11 7 14];
```

MATLAB's **colon** notation provides an easy way to extract a range of elements from `v`.

```
v(3:7) % Extract the third through the seventh elements
ans =
     9     4     2    11     7
```

The special **end** operator is an easy short-hand way to refer to the last element of `v`.

```
v(end) % Extract the last element
ans =
    14
```

The **end** operator can be used in a range.

```
v(5:end) % Extract the fifth through the last elements
ans =
     2    11     7    14
```

You can even do arithmetic using **end**.

```
v(2:end-1) % Extract the second through the next-to-last elements
ans =
     5     9     4     2    11     7
```

Indexing Vectors (3)

```
v = [16 5 9 4 2 11 7 14];
```

Combine the **colon** operator and **end** to achieve a variety of effects, such as extracting every k- th element or flipping the entire vector.

```
v(1:2:end) % Extract all the odd elements
```

```
ans =  
16 9 2 7
```

```
v(end:-1:1) % Reverse the order of elements
```

```
ans =  
14 7 11 2 4 9 5 16
```

By using an indexing expression on the left side of the equal sign, you can **replace** certain elements of the vector.

```
v([2 3 4]) = [10 15 20] % Replace some elements of v
```

```
v =  
16 10 15 20 2 11 7 14
```

Indexing Matrices with two subscripts

Now consider indexing into a matrix. We'll use a magic square for our experiments.

```
A = magic(4)
A =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

Most often, indexing in matrices is done using two subscripts - one for the rows and one for the columns. The simplest form just picks out a single element.

```
A(2,4) % Extract the element in row 2, column 4
ans =
     8
```

More generally, one or both of the row and column subscripts can be vectors.

```
A(2:4,1:2)
ans =
     5    11
     9     7
     4    14
```

Indexing Matrices with two subscripts

A single ":" in a subscript position is short-hand notation for "1:end" and is often used to select entire rows or columns.

```
A(3,:) % Extract third row
```

```
ans =
```

```
9 7 6 12
```

```
A(:,end) % Extract last column
```

```
ans =
```

```
13  
8  
12  
1
```

```
A = magic(4)
```

```
A =
```

16	2	3	13
5	11	10	8
9	7	6	12
4	14	15	1

There is often confusion over how to **select scattered elements from a matrix**. For example, suppose you want to extract the (2,1), (3,2), and (4,4) elements from A? The expression `A([2 3 4], [1 2 4])` won't do what you want. This diagram illustrates how two-subscript indexing works.

Indexing Matrices (linear Indexing)

	column 1	column 2		column 4
	16	2	3	13
row 2	5	11	10	8
row 3	9	7	6	12
row 4	4	14	15	1

A

5	11	8
9	7	12
4	14	1

A([2 3 4], [1 2 4])

There is a different possibility to select the elements of the matrix, thinking at the matrix like a **big column** vector and using the *linear indexing*.

Indexing Matrices (linear Indexing)

When you index into the matrix **A** using only one subscript, MATLAB treats **A** as if its elements were strung out in a long column vector, by going down the columns consecutively, as in:

Implicit
column
vector



16
5
9
4
2
11
.
.
-
8
12
1
.
.

1 16	5 2	9 3	13 13
2 5	6 11	10 10	14 8
3 9	7 7	11 6	15 12
4 4	8 14	12 15	16 1

A

The linear index of each element is shown in the upper left.

Indexing Matrices (linear Indexing)

The expression `A(14)` simply extracts the 14th element of the implicit column vector. Indexing into a matrix with a single subscript in this way is often called **linear indexing**.

From the diagram you can see that `A(14)` is the same as `A(2,4)`.

The single subscript can be a vector containing more than one linear index, as in:

```
A([6 12 15])  
ans =  
    11    15    12
```

Consider again the problem of extracting just the (2,1), (3,2), and (4,4) elements of `A`. You

can use linear indexing to extract those elements:

```
A([2 7 16])  
ans =  
     5     7     1
```

1 16	5 2	9 3	13 13
2 5	6 11	10 10	14 8
3 9	7 7	11 6	15 12
4 4	8 14	12 15	16 1

A

Indexing Matrices (linear Indexing)

That's easy to see for this example, but how do you compute linear indices in general?

MATLAB provides a function called **sub2ind** that converts from row and column subscripts to linear indices.

You can use it to extract the desired elements (2,1), (3,2), and (4,4) elements of A.

```
idx = sub2ind(size(A), [2 3 4],  
[1 2 4])
```

```
ans =  
  
2 7 16
```

```
A(idx)
```

```
ans =  
  
5 7 1
```

```
>> help sub2ind
```

SUB2IND Linear index from multiple subscripts.

SUB2IND is used to determine the equivalent single index corresponding to a given set of subscript values.

IND = SUB2IND(SIZ,I,J) returns the linear index equivalent to the row and column subscripts in the arrays I and J for an matrix of size SIZ.

Adding / Deleting elements

Can you delete a row or a column!

```
>> A=[1 2 -6; 4 -4 5]
```

A =

```
1     2    -6
4    -4     5
```

```
>> A(:,2)= []
```

A =

```
1    -6
4     5
```

BUT you cannot delete a single element!

```
>> A(2,1)= []
```

??? Indexed empty matrix assignment is not allowed.

Differently if you add one element out of the matrix dimensions:

A(1,8)=1

A =

```
1    -6     0     0     0     0     0     1
4     0     0     0     0     0     0     0
```

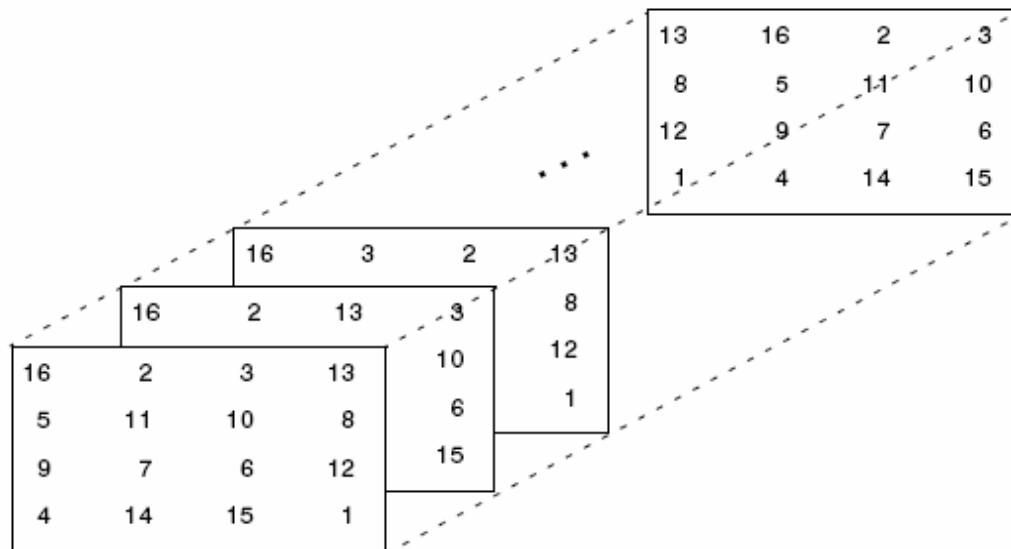
It is called matrix expansion

Multidimensional Arrays

Multidimensional arrays in MATLAB are arrays with more than two subscripts. One way of creating a multidimensional array is by calling `zeros`, `ones`, `rand`, or `randn` with more than two arguments. For example,

```
R = randn (3,4,5);
```

creates a 3-by-4-by-5 array with a total of $3 \times 4 \times 5 = 60$ normally distributed random elements.



In theory you can create vectors with more than 3 dimensions.

```
A = rand(2,2,2,2,2,2);
```

Generating Matrices

Bracket constructions are suitable only for very small matrices. For larger ones, there are many useful functions:

<code>eye</code>	Identity matrix	<code>>> eye(10)</code>
<code>zeros</code>	All zeros	<code>>> zeros(10)</code>
<code>ones</code>	All ones	<code>....</code>
<code>diag</code>	Diagonal matrix (or, extract a diagonal)	
<code>toeplitz</code>	Constant on each diagonal	
<code>triu</code>	Upper triangle	
<code>tril</code>	Lower triangle	
<code>rand</code>	Uniformly distributed random elements	
<code>randn</code>	Normally distributed random elements	
<code>linspace</code>	Evenly spaced entries	
<code>repmat</code>	Duplicate vector across rows or columns	

To see more
details on these
functions, use the
help

For example, **zeros (m,n)** produces an m-by-n matrix of zeros and **zeros (n)** produces an n-by-n one. If A is a matrix, then **zeros (size (A))** produces a matrix of zeros having the same size as A.

If x is a vector, **diag (x)** is the diagonal matrix with x down the diagonal; if A is a square matrix, then **diag (A)** is a vector consisting of the diagonal of A. What is **diag (diag (A))**?

Common scalar/vector functions

Scalar functions

Certain MATLAB functions operate essentially on scalars, but operate element-wise when applied to a matrix. The most common such functions are

sin	asin	exp	abs	round
cos	acos	log (natural log)	sqrt	floor
tan	atan	sign	ceil	

Vector functions

Other MATLAB functions operate essentially on a vector (row or column), but act on an m-by-n matrix ($m \geq 2$) in a column-by-column fashion to produce a row vector containing the results of their application to each column. Row-by-row action can be obtained by using the transpose; for example, `mean (A')'`. A few of these functions are

max	sum	median	any	→ e.g. Help sum
min	prod	mean	all	
sort	std	cumsum	cumprod	

For example, the maximum entry in a matrix A is given by `max (max (A))` rather than `Max (A)`. (e.g.....)

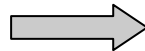
Reshaping Matrices

$B = \text{repmat}(A, m, n)$. Duplicate vector or matrices across rows or columns

```
>> A = rand(6,1)
```

A =

0.23
0.24
0.05
0.08
0.64
0.19



```
>> A = repmat(A,1,5)
```

A =

0.23	0.23	0.23	0.23	0.23
0.24	0.24	0.24	0.24	0.24
0.05	0.05	0.05	0.05	0.05
0.08	0.08	0.08	0.08	0.08
0.64	0.64	0.64	0.64	0.64
0.19	0.19	0.19	0.19	0.19

$B = \text{Reshape}(X, M, N)$. Returns the M-by-N matrix whose elements are taken columnwise from X. An error results if X does not have $M*N$ elements.

```
>> A = rand(6,1)
```

A =

0.27
0.25
0.87
0.23
0.80
0.91

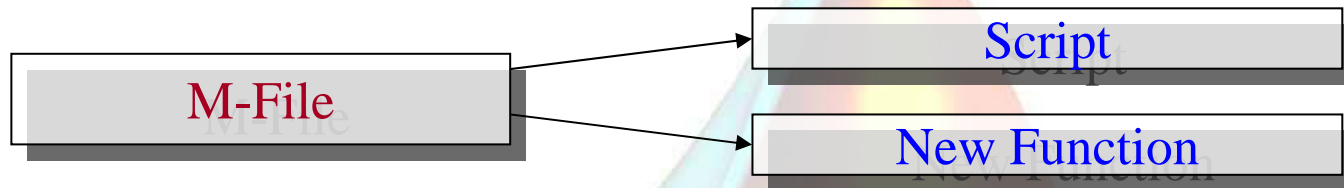


```
>> A = reshape(A,3,2)
```

A =

0.27	0.23
0.25	0.80
0.87	0.91

Command Window Vs. Program Editor (Scripts and functions)



➤ Working in the command window has some drawback:

- *The immediate execution of the code can cause **error messages** are printed in the screen*
- *Not using semicolon at the end of the code gives results displayed in the screen*

This is a problem if you want save the work (program) to use it next times. You have to copy only the rows of the code without error messages or results, and paste it in another place. BUT it is too much time consuming.

This operation is feasible also using the command **diary on / diary off**, BUT the problem remains.

- ✓ The solution is working using the **Program Editor**, a pretty good editor that is tightly integrated into the environment. However, you are free to use any text editor.
It allows you to save the work in a **M-File**. An **M-file** is a regular text file containing MATLAB commands, saved with the filename extension **.m**.

The program in an M-File could be run through the Program Editor (pressing the **F5** key or using the **run** button. You must save the program before you can run it.

All the programming rules valid in the command window are the same for the program editor

Command Window Vs. Program Editor (Scripts and functions)

Script

A program created to solve a specific problem

Simply type in the name of the file (without the extension) in order to run it

New Function

A program created following a **specific format**, callable in other scripts or functions

Functions are the main way to extend the capabilities of MATLAB.

Note : working in the command window or using the program editor are not two incompatible ways to built a program.

The typical use of the command is for short computations, BUT is also a crucial support when you write the program in the program editor. You can test in the command if a piece of code is correct and gives to you the expected result before you put it in the script or function.

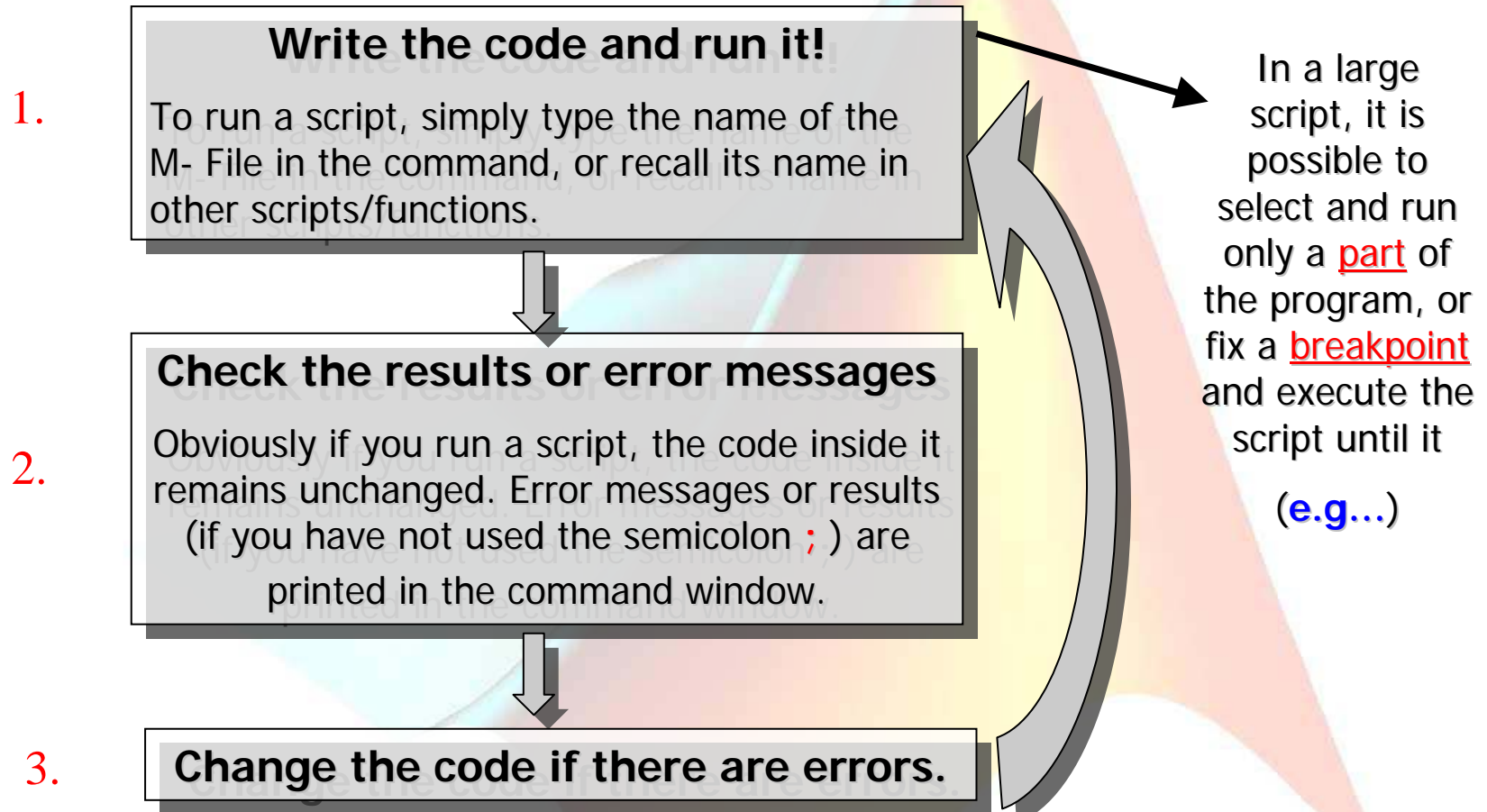
To open a new M-File
File > New > M-file

To open an old M-File
File > New > M-file

or

(Double-click on the file (**name.m**) in current directory)

Steps creating a script

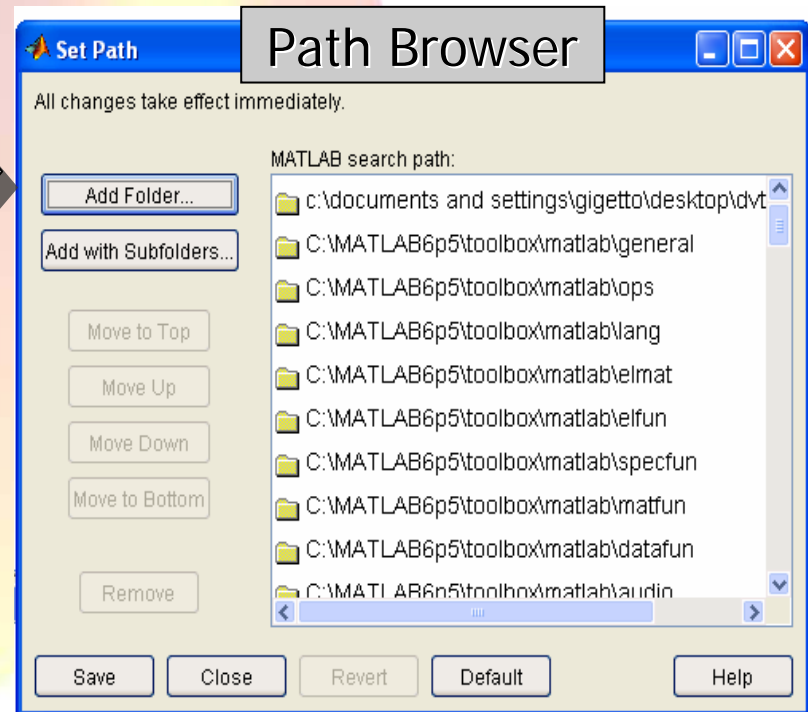


Changing the Work directory

- As said before, you can change the work directory, browsing from the panel *Current Directory*. If you do it, remember that you lose the property of the default directory "work". It means that MATLAB has a list of default folders, in which it searches for functions and scripts. The folder "work" is by default in that search list. But a new folder (e.g. in the Desktop) is not in the list until you haven't specified it.

To do it: **File > Set Path > Add Folder**

After that it is possible to work in a different new folder, and call functions and scripts saved in the above folder, simply typing their names in the command window.



Getting Help (learning by doing)

MATLAB is a huge package. You can't learn everything about it at once, or always remember how you have done things before. It is essential that you learn how to teach yourself more using the online help. In MATLAB the **Help** is not a tool to use only when something is gone wrong. You have to use it to learn working in MATLAB by yourself.

- The lookfor command
- The help command
- The help window

>> lookfor name
>> help "name of the function"
>> helpwin or Help > MATLAB Help

>> **lookfor** scholes

BLSDELTA Black-Scholes sensitivity to underlying price change.

BLSGAMMA Black-Scholes sensitivity to underlying delta change.

BLSIMPV Black-Scholes implied volatility.

BLSLAMBDA Black-Scholes elasticity.

BLSPRICE Black-Scholes put and call pricing.

.....

>> **help** blsprice

BLSPRICE Black-Scholes put and call pricing.

[CALL,PUT] = BLSPRICE(SO,X,R,T,SIG,Q) returns the value of call and put options using the Black-Scholes pricing formula. SO is the current asset price, X is the exercise price, R is the risk-free interest rate, T is the time to maturity of the option in years, SIG is the standard deviation of the annualized continuously compounded rate of return of the asset (also known as volatility), and Q is the dividend rate of the asset.

The default Q is 0.

.....

Getting Help

Using the command `help` followed by the name of a function gives to you a short explanation of what the function does, and which are the inputs and outputs. The short explanation is shown in the command window.

E.g.: `help exprnd`

```
>> help exprnd
```

EXPRND Random matrices from exponential distribution.

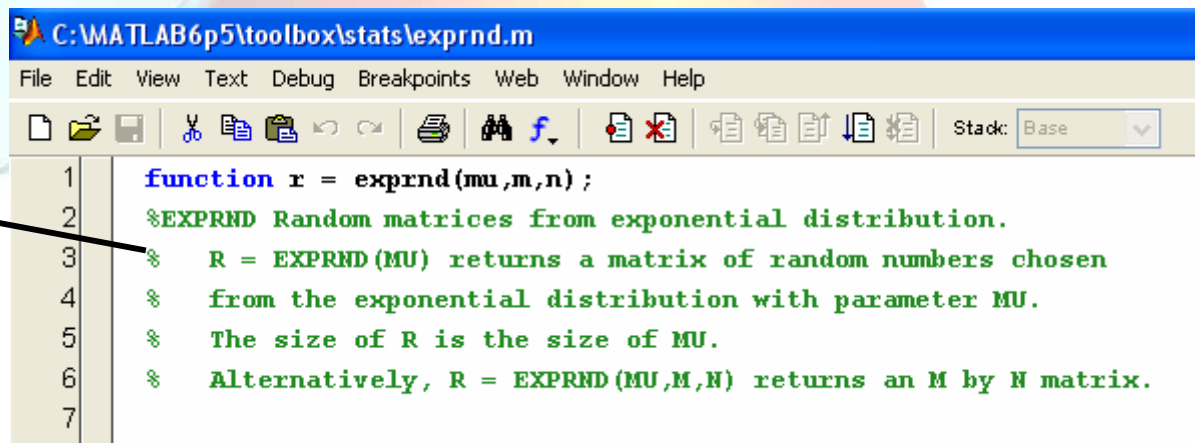
`R = EXPRND(MU)` returns a matrix of random numbers chosen from the exponential distribution with parameter MU.
The size of R is the size of MU.

Alternatively, `R = EXPRND(MU,M,N)` returns an M by N matrix.

If you open the M-file "`exprnd.m`" (the name of the file is ever the same of the function name) you can see that the explanation text printed in the screen, is the text put at the beginning of the code.

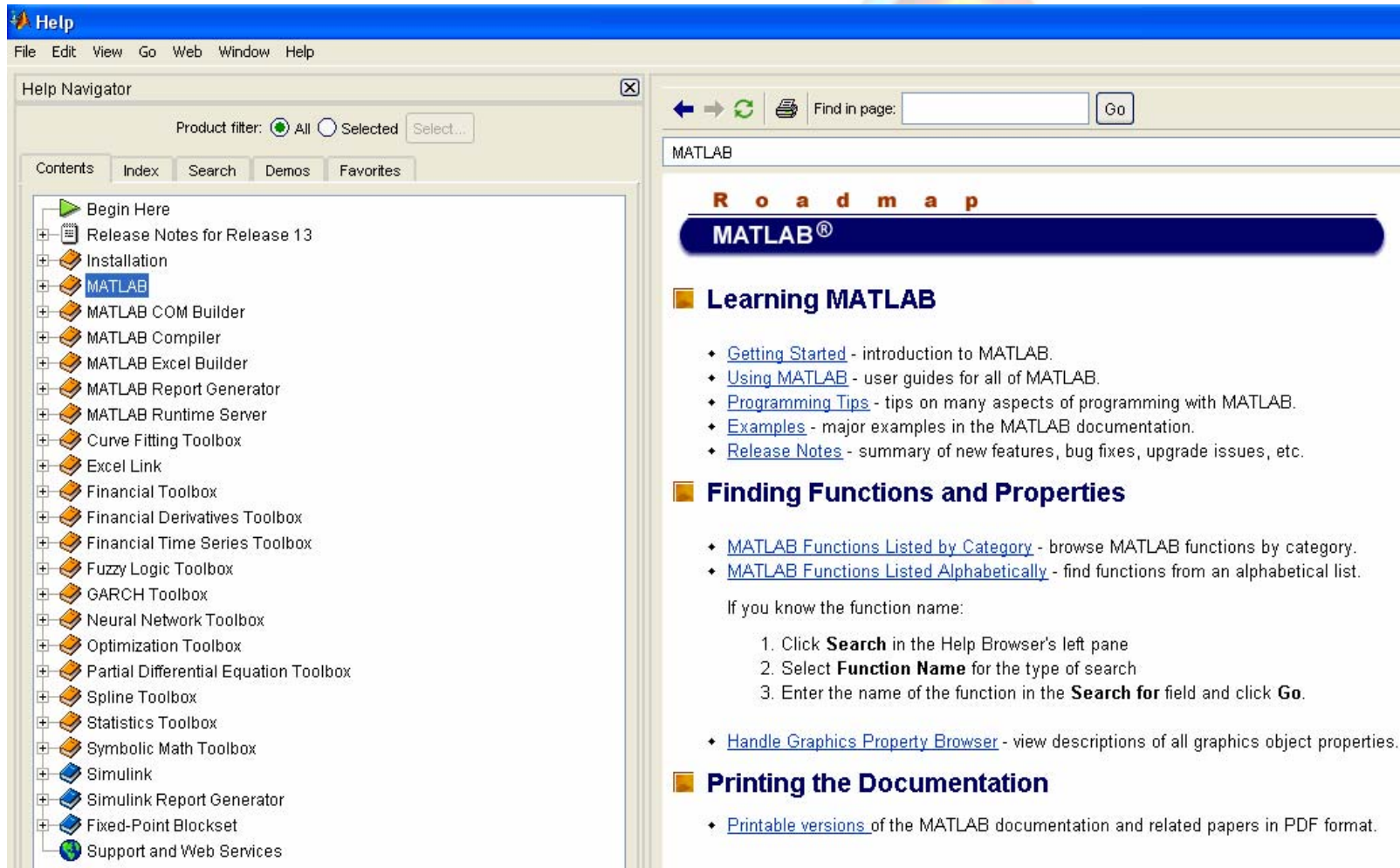
The code after percent sign **%** is a comment and is not run.

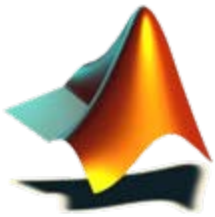
Any text on the same line after a percent sign is ignored



```
C:\MATLAB6p5\toolbox\stats\exprnd.m
File Edit View Text Debug Breakpoints Web Window Help
[Icons] Stack: Base
1  function r = exprnd(mu,m,n) ;
2  %EXPRND Random matrices from exponential distribution.
3  % R = EXPRND(MU) returns a matrix of random numbers chosen
4  % from the exponential distribution with parameter MU.
5  % The size of R is the size of MU.
6  % Alternatively, R = EXPRND(MU,M,N) returns an M by N matrix.
7
```

Help window





Loops, IF statement, logical and relational operators

Relational and logical operators

The relational operators in MATLAB are

<	less than
>	greater than
<=	less than or equal
>=	greater than or equal
==	equal
~=	not equal.

Note that "=" is used in an **assignment statement** while "==" is used in a relation.

Relations may be connected or quantified by the **logical operators**

When applied to scalars, a relation is actually the scalar 1 or 0 depending on whether the relation is true or false.

(e.g. `3 < 5`, `3 > 5`, `3 == 5`, and `3 == 3`.)

When applied to matrices of the same size, a relation is a matrix of 0's and 1's giving the value of the relation between corresponding entries.

(e.g. `a = rand(5)`, `b = triu(a)`, `a == b`)

To evaluate logical expressions

&& and

|| or

~ not

Working element-by-element on arrays

& and

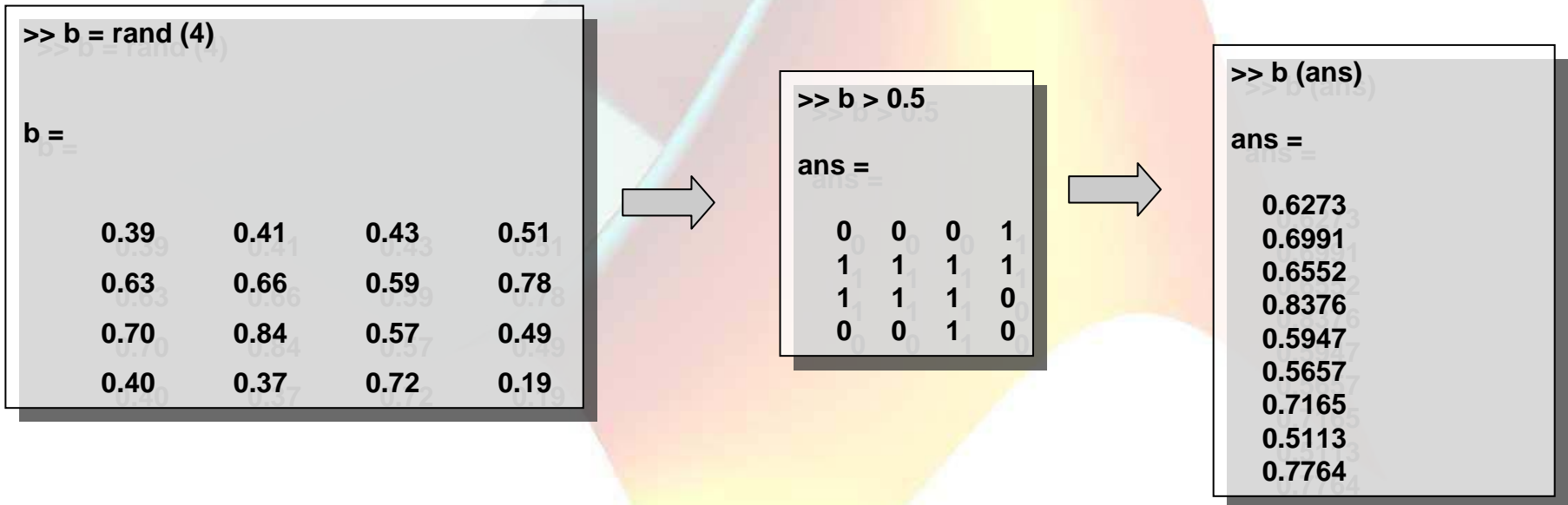
| or

~ not

Relational indexing

A different kind of indexing (really powerful in many cases) is **logical indexing**.

Logical indices usually arise from a **relational operator**. The result of applying a relational operator is a **logical array**, whose elements are 0 and 1 with interpretation as “false” and “true.” **Using a logical array as an index returns those values where the index is 1** (in the single-index sense above).



The find command

MATLAB has a FIND command that is extensively used. It returns the indices of non-zero elements of a matrix, and it is FAST.

```
M = magic(3)
```

```
M =
```

```
8 1 6
```

```
3 5 7
```

```
4 9 2
```

```
ind = find(M > 4)
```

```
ind =
```

```
1
```

```
5
```

```
6
```

```
7
```

```
8
```

(M > 4) produces a matrix of 1's and 0's; 1 if the inequality is true for that element and 0 if it isn't. FIND then returns the indices of all the 1's.

Note: The indices are in the linear format (linear indexing)

For loops

This illustrates the most common type of **for** loop:

```
>> f = [1 1];  
>> for n = 3:10  
    f(n) = f(n-1) + f(n-2);  
end
```

You can have as many statements as you like in the body of the loop. The value of the index **n** will change from 3 to 10, with an execution of the body after each assignment. But remember that 3:10 is really just a row vector. In fact, you can use any row vector in a **for** loop, not just one created by a colon operator. For example,

```
>> x = 1:100; s = 0;  
>> for j = find(isprime(x))  
    s = s + x(j);  
end
```

This finds the sum of all primes less than 100.

Typically, the **if** statement is used inside a **for** loop, to verify a conditional statement.

If statement

To write programs of any complexity you need to be able to use iteration and decision making. These elements are available in MATLAB much like they are in any other major language. For decisions, there is the **if** statement, and for iterations there are **for** loops.

Syntax

```
if expression (conditional statement)
    statements;
end
```

The conditions for **if** statements may involve the **relational operators** seen before.

```
if    expression1
    statements1;
elseif expression2
    statements2;
else
    statements3;
end
```

Vectorization

```
M = magic(3)
M =
     8     1     6
     3     5     7
     4     9     2
```

and you want to negate every element greater than 4. The way you'd do this in conventional (**scalar**) languages is,

```
for i=1:3,
    for j=1:3,
        if (M(i,j) > 4),
            M(i,j) = - M(i,j);
        end
    end
end
```

Alternatively

```
ind = find (M > 4);
M(ind)=-M(ind);
```

methods on a 300x300 matrix

For loop method:

elapsed_time = 23.4956

FIND method:

elapsed_time = 2.1153

Optimization

Optimization Toolbox Minimization

Fgoalattain

Multiobjective goal attainment

Fminbnd

Scalar nonlinear minimization with bounds

Fmincon

Constrained nonlinear minimization

Fminimax

Minimax optimization

fminsearch,fminunc

Unconstrained nonlinear minimization

Fseminf

Semi-infinite minimization

Linprog

Linear programming

Quadprog

Quadratic programming

Utility

Optimget

Get optimization options parameter values

Optimset

Create or edit optimization options parameter structure

OPTDEMO Tutorial for Optimization Toolbox



The function analyzed is:

$$f(x_1, x_2) = e^{x_1} \cdot (4x_1^2 + 2x_2^2 + 4x_1x_2 + 2x_2 + 1)$$

- (1) unconstrained optimization example
- (2) constrained optimization example
- (3) constrained example using gradients
- (4) bounded example
- (5) equality constrained example
- (6) unconstrained example using user-supplied tolerances