Programming with MATLAB

Clodomiro Ferreira **Aleksei Netsunajev**

EUI

February 10, 2011

Preliminaries (I)

- **MATLAB knowledge we assume:** very basic - basic operations and matrix manipulation, feeling confortable with the main MATLAB windows...
- **Objective of the tutorial:** Allowing you to feel confortable when solving computational and numerical problems in MATLAB

Preliminaries (II)

Levels of programming:

- Mathematica / Maple
- MATLAB /Gauss
- Fortran / C++

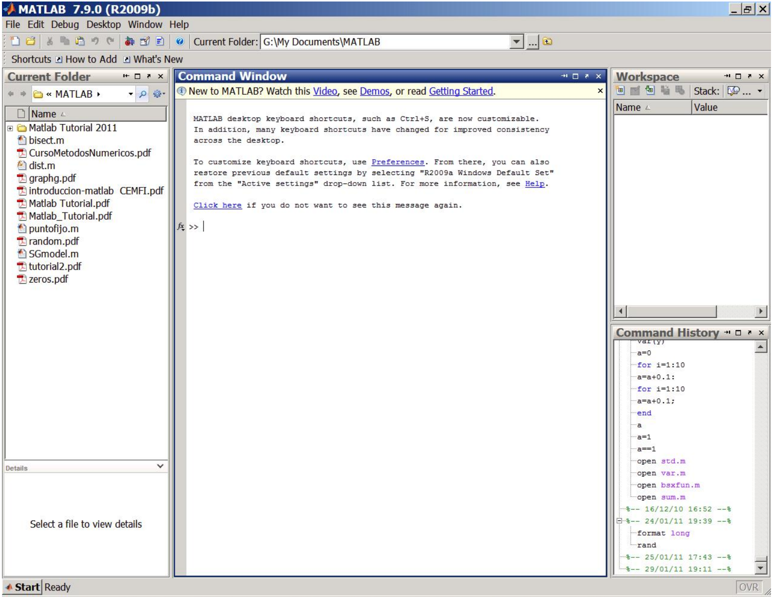
Tentative Outline

1. Before we begin
2. M-files, Scripts and Functions
3. Control of Flow: if, for, while, ...
4. Programming: general issues
5. Graphics in MATLAB
6. MATLAB Help
7. Applications

Working environment

- Command Window
- Command History
- Workspace
- Current Directory

Working environment



Variables in MATLAB: Matrices or Arrays

- (Most) objects you define are understood as n-dimensional arrays by MATLAB
- You **do** need to: define a *name* for the object.
- You **dont** need to: define the *dimensions* of the object.
- Example: write in the *command window*

$$a = 3$$

MATLAB creates a 1x1 **array** named *a*. Check: write

$$size(a)$$

- **Tip**: use the function `size(·)`!

Defining Matrices *explicitly*

More...

- $M = [1,2;3,4;5,6]$ is a 3x2 matrix

$$M = \begin{matrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{matrix}$$

- $M = [low : step : high]$ creates a **row** vector with first element `low`, last element `high` and distance between elements `step`
- $M =$ `linspace(0,1,5)` creates a row vector with 5 elements

$$M = (0, 0.25, 0.5, 0.75, 1)$$

- **Remember:** use the semi-colon ";" after a command in order to tell MATLAB not to show output on the *Command window*.

Accessing sections of a Matrix

More...

$$M = \begin{matrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{matrix}$$

- Accessing element (i, j): $M(i, j)$. Type

$$b = M(1, 2)$$

  b = 2

- Accessing row i: $M(i, :)$
- Accessing a particular **range**: $M(i_1 : i_2, j_1 : j_2)$

Useful Matrices and operations

More...

- $N_1 \times N_2$ matrix of **ones**: `ones(`N_1, N_2`)`
- $N_1 \times N_2$ matrix of **zeros**: `zeros(`N_1, N_2`)`
- $N_1 \times N_2$ **identity** matrix : `eye(`N_1, N_2`)`
- If M is an $n \times m$ matrix, many common operations are available as MATLAB commands: `inv`$(\cdot)$, `det`$(\cdot)$, `eig`$(\cdot)$

Useful 1: MATLAB allows you to work on an "*element by element*" basis. Just add a "*dot*" n front of the operator:

$$M.^{*}M = \begin{matrix} 1 & 4 \\ 9 & 16 \\ 25 & 36 \end{matrix}$$

Useful 2: *Multidimensional arrays*. MATLAB allows you to create *arrays* with more than two dimensions. For example, `A=zeros(2,2,3)` creates a "cube" formed by three 2×2 arrays.

M-files & Scripts: writing your own programs

- Main tool for writing code in MATLAB.
- For simple problems, entering your requests at the Matlab prompt is fast and efficient. However, as the number of commands increases typing the commands over and over at the Matlab prompt becomes tedious.
- Similar advantages to a `.do` file in Stata.
- All built-in commands (i.e. `mean(.)`, `sqrt(.)`, `inv(.)`, etc) are `.m` files.
- Two types of `.m` files:
  - **Script** files: do not take imput or retur/output arguments
  - **function** files: may take imput / return arguments

M-files & Scripts: writing your own programs

- In order to create and run an .m file, you need to:
  - `File`→`New` →`M-file`. File with a `.m` extension.
  - Give it a name. Be sure the name is not an existing function!!
    >> help clodo
    clodo.m not found.
  - Write your program / instructions.
    - ⋆ Inside an `.m` file, you can "call" other `.m` files.
    - ⋆ Write **comments** on your program!
  - Save it on the **current directory** (cd).
  - "call it": type on the *Command Window* clodo (or `run` clodo)
- Variables and output created when running the .m file will be stored on the *Workspace*.

Script File: simple example

```
This program generates pseudo-random sequence of 0 and 1
clc
clear all
L = 10 ;
x = rand(1,L) ;
y = round(x) ;
z = sum(y,2) ;
y
z
```

Script File: simple example

```
clc         % "Clears" the command window
clear all % Eliminates alll previous variables stored in the workspace
% This program will generate a "quasi"random sequence of 0 and 1

%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%        PARAMETERS
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
L=10; %Length of sequence

%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%        COMPUTATIONS
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% First we create a vector of uniformly distributed pseudo random numbers
x=rand(1,L);
% Now we "round" such numbers
y=round(x);
y %shows vector y in the command window
%finally, lets sum all elements of y
z=sum(y,2);
z %show sum

```

Built-in & own Functions

- **Functions**: `.m` files that can accept imput arguments and return output arguments.
  - Built-in: functions already existing in MATLAB. Example: `inv(.)`, `regress(.)`, `plot(.)`, etc.
  - Own functions: functions created by **you**

Built-in function: regress

Type `help regress`

Statistics Toolbox

regress

Multiple linear regression

Syntax

```
b = regress(y,X)
[b,bint,r,rint,stats] = regress(y,X)
[b,bint,r,rint,stats] = regress(y,X,alpha)
```

Description

`b = regress(y,X)` returns the least squares fit of `y` on `X` by solving the linear model

$$y = X\beta + \varepsilon$$
$$\varepsilon \sim N(0, \sigma^2 I)$$

for β, where:

- y is an n-by-1 vector of observations
- X is an n-by-p matrix of regressors
- β is a p-by-1 vector of parameters
- ε is an n-by-1 vector of random disturbances

`[b,bint,r,rint,stats] = regress(y,X)` returns an estimate of β in `b`, a 95% confidence interval for β in the p-by-2 vector `bint`. The residuals are returned in `r` and a 95% confidence interval for each residual is returned in the n-by-2 vector `rint`. The vector stats contains the R^2 statistic along with the F and p values for the regression.

Built-in function: `regress`

Two things to note: ▸ More useful functions...

1. Imput arguments (X, y), can have ***different*** dimensions
2. A function can have one or more imput arguments (with a maximum) and one or more output arguments.
   1. `b=regress(y,X)`
   2. [b,bint,r,rint,stats]=`regress(y,X)`

Own Functions

Useful when we want to automatize a particular set of operations which require imput arguments from another set of operations.

General structure:

```
function f = myfun(x,y, ...)
commands
...
f = expression;
```

or with more output arguments

```
function [f1,f2] = myfun(x,y, ...)
commands
...
f1 = expression;
f2 = expression;
```

Own Functions: key features

- All function `.m` files start with the command `function`
- `f` is the output. Can be replaced by [f,z,w,...], i.e. more than one outputs
- Unless you specify it explicitely (using `local` and `global` commands, variables (and their names) used within the function are are not stored in the workspace, and are just recognized inside this function.
- ***Key:*** You need to save the `.m` file with the name **myfun**
- After you specified a set of commands, you need to explicitely specify the output; hence the last line `f=...`

Own Functions: example 1

- Lets create a function that evaluates the expression

$$f(x,y) = x^2 + y^3 + \frac{\sqrt{x+y}}{2}$$

```
function result = funct1 ( x , y ) imputs:  x,y ; output:
result
result = x^2 + y^3 + sqrt(x+y) / 2 ;
```

- Then, if we type in the Command Window (or we call it from another .m file) funct1(1,2) we get
  ans =
  9.8660

Own Functions: example 1

More remarks:

- The name of the imputs, x and y, are only valid within the function.
- Usually, such imputs come fro previous calculations or parameters defined within a "main" program.

Control of Flow

- MATLAB has four basic decives a programmer can use to control de **flow** of commands:
  - **for** loops
  - **if-else-end** constructions
  - **while** loops
  - **switch-case** constructions

Repeating with for loops

- For loop repeats a group of statements a fixed, predetermined number of times.

General structure:

```
for k=array
...
end
```

Simple example

```
x=zeros(1,5) row of zeros to store
for n=1:5
x(n)=n^2;
end
```

x= 1 4 9 16 25

You can *nest* `for` loops...

`for` loops can be **nested**:

- `for` i=1:4
  - `for` j=1:4
    - x(i,j)= i * j ;
  - `end`
- `end`

this will generate the following matrix:

$$x = \begin{matrix} 1 & 2 & 3 & 4 \\ 2 & 4 & 6 & 8 \\ 3 & 6 & 9 & 12 \\ 4 & 8 & 12 & 16 \end{matrix}$$

... but should avoid nesting whenever possible!!

- MATLAB comparative advantage: **vectorization** and working with matrices.
- *Nested* loops are **much slower** than working with vectors
- the previous nested loop can be simplified:

```
i=1:4; row vector 1 2 3 4
x=i ' * i ; vector multiplication. Careful with dimensions
```

- Also: always **predefine** the matrix where you want to store

Repeating with `while` loops

- This loop is used when the programmer **does not know the number of repetitions a priori**

General structure:

d = d0 Initialize variable d
`while` *expression with* ***d***
...
d = ... update d
`end`

- Useful for iterations on recursive probems...

Repeating with while loops: example

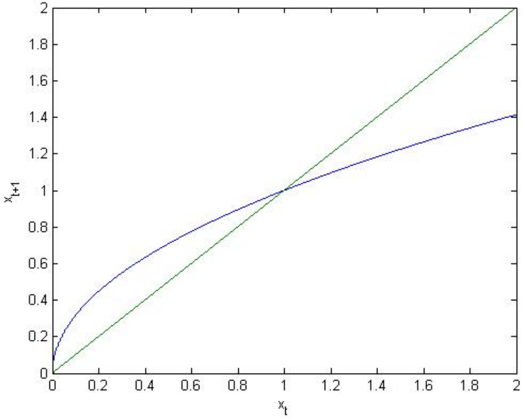
Simple fixed point problem:

```
x0=0.5; initial value
d=1 ; distance. Will be updated
tol=0.0001 ; tolerance value
while d>tol
    x1 = sqrt ( x0 ) ;
    d = abs ( x1-x0 ) ; update of distance
    x0 = x1
end
```

x1=0.9999
d=8.4602e-005

Repeating with `while` loops: example

Figure: Fixed point



`if-else-end` constructions

- Do some operations ***if*** some conditions hold.

General structure:

- One alternative

  if *expression*

  ...

  end

- More than one alternative

  if *expression 1*

  ...

  elseif *expression 2*

  ...

  else

  ...

  end

Relational and logical operators

Operator	Description
Relational	
>, <	greater / lower than
>=, <=	greater / less or equal
==	equal
~=	not equal
Logical	
&	and
|	or
~	not

Example: Simulating a Markov chain w/ 2 states

```
T=10; no. of periods
P=[0.7,0.3;0.4,0.6]; transition matrix
Y=[0.9,1.1]; possible values of state
u=rand(1,T); draws from a U(0,1)
z=zeros(1,T);
t=1; initialization
z(1)=Y(1); initialization
index0=1; index for current state
```

- while t<T
  - if u(t)<P(index0,1)
    z(t)=Y(1);
    index0=1;
    else
    z(t)=Y(2);
    index0=2;
    end

  t=t+1;
  end

The `switch-case` construction

- Switch compares the input expression to each case value. Once the match is found it executes the associated commands.
- For most practical cases, it achieves similar results to `if-elseif-else` constructions

General structure:

switch *expression* scalar or string
case *value1* executes if expression = value1
commands...
case *value2*
commands...
...
otherwise
commands
end

Control of flow: programming tips

- Try to program "inside out": start with the inner section of your code, check it produces the desired results, and proceed towards the "outer" loops. In this way, you keep track of each step and possible errors.
- "Ask" MATLAB to **print intermediate results** / variable values. This is a good way to know exactly what is going on inside your code.
- **Check the *Workspace* window**: sometimes, either the dimensions of your matrices are not the ones you though... or matrices are just empty!!!

(Some) "common" error messages

- ???Error using ==> minus
  Matrix dimensions must agree. : Often it is an indexing mistake that causes the terms to be of different size.
- ??? Error using ==> mtimes
  Inner matrix dimensions must agree.: Note the difference between this error and the previous one. This error often occurs because of indexing issues OR because you meant to use componentwise multiplication but forgot the dot.
- Error: Unbalanced or misused parentheses or brackets. : for a complex expression you have to go through it very carefully to find your typo.
- ??? Error using ==>
  Too many input arguments. : Input arguments must be in a format expected by the function. This will be very function-specific.

Logical Addressing in MATLAB

- You can solve some tricky problems using some **logical addressing**.
- Two useful functions / operations:
  - `find(.)` finds **indices** of non-zero elements in an array
  - $(\mathtt{\mathit{expression}})$ acts as an **indicator** function: it takes value $= 1$ if *expression* holds

Logical Addressing: example

$$A = \begin{matrix} 1 & 1 & 3 \\ 5 & 9 & 2 \\ 4 & 4 & 6 \end{matrix} \qquad x = \texttt{rand}(1,3) = 0.1576\ 0.9706\ 0.9572$$

1. Find the indices where A>=4: ind1 = $\texttt{find}(A >= 4)$
   ind1 = 2 3 5 6 9
2. Operate over values of x that satisfy certain condition:

$$z = (\mathrm{x.\hat{}}2) \,.*\, (x > 0.2)$$

$z = (0, 0.9421, 0.9162)$

Basic commands

Be careful with function *sqrt*() using matrices. Consider example:

$$A = \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix};$$

$$B = sqrt(A)$$

$$B = \begin{bmatrix} 1.4142 & 1.4142 \\ 1.4142 & 1.4142 \end{bmatrix};$$

$$B' * B = \begin{bmatrix} 4 & 4 \\ 4 & 4 \end{bmatrix}$$

$$C = chol(A);$$

$$C' * C = \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}$$

Programming

A variable is a tag that you assign to a value while that value remains in memory. You refer to the value using the the tag.

- You do not need to type or declare variables.
- MATLAB variable names must begin with a letter. MATLAB is case sensitive, so A and a are two different variables!
- Do not name a variable using a reserved names, such as $i, j, mode, char, size$ and $path$.

Programming. Structures

Structures are multidimensional MATLAB arrays. This is very much like a database entity. Structures are useful to group variables.

Let structure consist of 3 variables: Student.name, Student.score, Student.grade. The whole structure could be an input for user-define function. It is convenient to use the structures when you have a lots of variables and you use your own functions. You won't need to pass all 3 variables to the function, but just the whole structure

See provided example.

Programming. Local variables

Each MATLAB function has its own variables. These are separate from those of other functions, and from those of the base workspace, hence they are called local.They 'live' only while the function is running.

Scripts do not have a separate workspace. They store variables in a workspace that is shared with the caller of the script. When called from the command line, they share the base workspace. When called from a function, they share that function's workspace. If you run a script that alters a variable that already exists in the workspace, that variable is overwritten by the script.

Programming. Global variables

If several functions, and the base workspace, all declare a particular name as a global variable, then they all share a single copy of that variable. Any assignment to that variable, in any function, is available to all the other functions declaring it as global.

Instead of using a global variable, you may want to pass the variable to other functions as an additional argument. In this way, you make sure that any shared access to the variable is intentional.

If you have to pass a number of additional variables, you can conveniently put them into a structure and pass it as one argument.

See provided example.

Programming. Debugging

Do you think your program is not producing the results that you expected? Then you can debug your program and see what's wrong.

The standard debug tool are the breakpoints. Set breakpoints to pause execution of your program so you can examine values of variables where you think the problem can be.

After setting breakpoints, run the file

Programming. Debugging

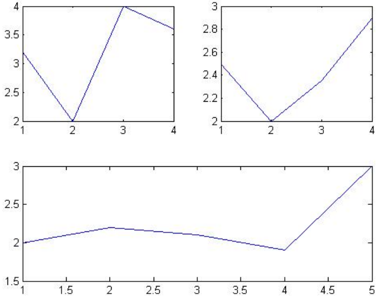
Then the Debug menu allows to:

- **Run** Commence execution of file and run until completion or until a breakpoint is met.
- **Go** Until Cursor Continue execution of file until the line where the cursor is positioned. Also available on the context menu.
- **Step** Execute the current line of the file.
- **Step** In Execute the current line of the file and, if the line is a call to another function, step into that function.
- **Continue** Resume execution of file until completion or until another breakpoint is encountered.
- **Step** Out After stepping in, run the rest of the called function or subfunction, leave the called function, and pause.
- **Exit** Debug Mode Exits debug mode.

2D Plots

There are many tools and ways for creating and editing your plots, both from the command line and by using the menus of Matlab (interface in the Figure window). It is possible to export your graph to nearly all conventional formats (.pcx, .bmp, .jpg, .pdf) via Save As option.

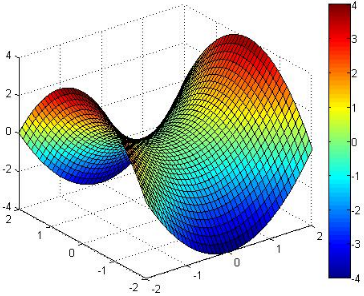
See provided example on plots and subplots.



3D Plots

The primary method to create the 3D plot is the surf command which is used in combination with the meshgrid command. Meshgrid creates a matrix of (x , y) points over which the surface is to be plotted.

See provided example on 3D plots.



MATLAB help

Matlab has the very user-friendly and extensive built-in and on-line help system. To access built-in help

- Type *help* in the Command Window
- Press F1
- Go to Help menu

Online user's guide is available at
http://www.mathworks.com/help/techdoc/matlab_product_page.html

Google the function you need end exploiting the web resourses available

Application 1: Bisection Algorithm

- We have a consumption saving problem with idiosyncratic uncertainty (but no aggregate uncertainty), borrowing constraints and no labor decision. The standard euler equation is given by

$$u^{'}(c_t) = \beta(1+r)E_t\left(u^{'}(c_{t+1})\right) \tag{1}$$

  and the period budget constraint

$$c_t = s_t w_t + (1+r)k_t - k_{t+1}$$

- Here, given no labor decisions and no aggregate uncertainty, w (labor wage) and r (net real interest rate) are time invariant. s_t is the current labor productivity state.
- (1) is usually non-linear. We wish to find the value of k_{t+1} that, for given $k_{t,}s_t$ solves the Euler equation.
- To do this, one strategy is using a **bisection algorithm**

Optimization toolbox: FMINCON

FMINCON is the function with the most features hence we base the example on it. Other optimization functions are less complex and work in a similar way.
The function is designed to find minimum of constrained nonlinear multivariate function:

$$min_x f(x) s.t. \left[\begin{array}{c} c(x) \leq 0 \\ ceq(x) = 0 \\ Ax \leq b \\ Aeq \cdot x = beq \\ lb \leq x \leq ub \end{array} \right.$$

where x, b, beq, lb, ub are vectors, A and Aeq are matrices, $c(x)$ and $ceq(x)$ are functions that return vectors, and $f(x)$ is a function that returns a scalar.

Optimization toolbox: FMINCON. Examples

Example 1. Find values of x that minimize $f(x) = -x_1 x_2 x_3$, starting at the point $x = [10; 10; 10]$, subject to the constraints: $0 \leq x_1 + 2x_2 + 2x_3 \leq 72$.

Example 2. Minimize log-likelihood function:

$$l(B, \Lambda_2, ..., , \Lambda_M) = T \log \det(B) + \frac{1}{2}\left(B'^{-1} B^{-1} \sum_{t=1}^{T} \xi_{1t|T} \hat{u}_t \hat{u}_t'\right)$$
$$+ \sum_{m=2}^{M} \left[\frac{T_m}{2} \log \det(\Lambda_m) + \frac{1}{2} tr \left(B'^{-1} \Lambda_m^{-1} B^{-1} \sum_{t=1}^{T} \xi_{mt|T} \hat{u}_t \hat{u}_t'\right)\right]$$

with respect to matrices B, Λ_m and taking other parameter as given, possibly subject to $B = \begin{bmatrix} * & 0 & 0 \\ & * & 0 \\ & * & * \end{bmatrix}$ and all elements of $diag(\Lambda_m)$ are ≥ 0.01.

Optimization toolbox: FSOLVE

The function *fsolve* is meant to solve system of nonlinear equations

Syntax:
$[X, fval] = fsolve(fun, x0, options)$
fsolve finds a root (zero) of a system of nonlinear equations.

Example:
Find a matrix x that satisfies the equation $xxx = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ starting at the point $strt0 = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$.

Defining matrices

Return

Input	Output	Comments
x = [1 2 3]	x = 1 2 3	row vector
x = [1;2;3]	x = 1 2 3	column vector
A = [1 2 3; 4 5 6]	A = 1 2 3 4 5 6	2 x 3 matrix

Accessing matrices

Return

Input	Output	Comments
$A = \begin{matrix} [1 & 2 & 3; \\ 4 & 5 & 6; \\ 7 & 8 & 9]; \end{matrix}$	supressed	create matrix
A(2,3)	ans = 6	element in 2nd row, 3rd col
A(:,3)	$\text{ans} = \begin{matrix} 3 \\ 6 \\ 9 \end{matrix}$	3rd col
A(2,:)	ans = 4 5 6	2nd row
A(1:2,2:3)	$\text{ans} = \begin{matrix} 2 & 3 \\ 5 & 6 \end{matrix}$	block

Special matrices

Return

Input	Output	Comments
x = zeros(2,4)	$x = \begin{matrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{matrix}$	2x4 matrix of zeros
x = ones(2,3)	$x = \begin{matrix} 1 & 1 & 1 \\ 1 & 1 & 1 \end{matrix}$	2x3 matrix of ones
A = eye(3)	$A = \begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{matrix}$	3x3 identity matrix

Useful Built-in functions (I)

- `error('error message')`: displays *error message* and abort function when a certain condition is satisfied
- `tic`: starts a time counter. t = `tic` assigns the current time to the variable t
- `toc`: this will display time elapsed since `tic` was called.
- `fprint`('*abc*'): prints text *abc* on the *Command Window*
- `size(.)` : rturns matrix / array dimensions
- `rand(n,m)`: generates an $n \times m$ matrix of pseudo-random numbers from a $U[0, 1]$
- `sort(X,dim)`: sorts elements of array X along dimension `dim`

Useful Built-in functions (II)

- `floor(x)`: rounds x towards minus infinity
- `ceil(x)`: rounds x towards plus infinity
- `round(x)`: rounds x towards nearest integer

Let x = [0.2234 , -1.4434 , 5.3789]. Then:

- `floor(x)` = [0 , -2 , 5].
- `ceil(x)` = [1 , -1 , 6].
- `round(x)` = [0 , -1 , 5].

(Pseudo-) Random Numbers in MATLAB (I)

Two built-in functions to generate pseudo-random numbers:

1. `rand(.)`; uniformly [0,1] distirbuted pseudo rn.
   1. a = `rand`; generates a scalar-random number
   2. A = `rand(n,m)` generates an nxm matrix of random numbers
2. `randn(.)`; (standard) normally distributed pseudo-rn
3. `rand('state',0)` or `randn('state',0)` useful to repeat a computation using the same sequence of pseudo- random numbers.

(Pseudo-) Random Numbers in MATLAB (II)

Return

Two alternative ways of generating normally distributed (pseudo) rn:

1. **probability integral transform property**: if U is distributed uniformly on (0,1), then $\Phi^{-1}(U)$ will have the standard normal distribution.
2. (approximation) **Central Limit Theorem**! Generate 12 $U \sim [0,1]$, add them up, and substract 6. (11th order polynomial approx. to the normal distribution)