Università della Svizzera italiana | Institute of Computing CI

Numerical Computing 2023

Introduction to MATLAB for Scientific Computing

Dr. Edoardo Vecchi

edoardo.vecchi@usi.ch

Lecture Overview

- This set of slides contains the material concerning lecture 1, which consists in an introduction to Matlab for scientific computing with some applications.
- Specifically, the lecture is organized in the following two parts:
  - We start by covering the basics of MATLAB, focusing our attention on all the relevant tools that can help you get immediately started. In particular, you can also find some friendly suggestions, which should prove useful to approach specific problems.
  - At the end of the lecture, you can find some exercises that can be solved by writing basic MATLAB programs. They aim at covering the basics of creating functions and scripts, plotting and conditional statements, while eventually reviewing basic probability concepts.
- Feel free to get in touch with me or the TAs should you have any doubts or concerns.

Overview

Why MATLAB?

Some of the reasons behind our choice of using MATLAB:

- No initialization or declaration of variables
- No compilation problems
- No particular constraints on variables naming[1]
- Many ways to produce nice-looking graphical outputs (useful for the project)
- Extensive numerical libraries and matrix-functions
- Fast vectorised algorithms
- Simple interface to other programming languages (e.g., C, C++, Fortran, Java)
- MATLAB is very simple to learn and use

[1] However, remember that MATLAB is case-sensitive.

Creating Scalars and Vectors

In order to assign the value 7 to variable `x`, we proceed as follows:

```
x = 7;
```

⋆ **Friendly Advice:** Notice that we have used the semi-column (;) to suppress the output in the command window. It is good practice to always do it, since printing all your results (e.g., in an iterative method) can have a negative impact on your code's performance.

In order to create a row vector `y` with 3 components, we use the square brackets, with the entries either separated by a space or a comma:

`y = [1 2 3];` or `y = [1, 2, 3];`

If `y` is a column vector, we need instead to separate the components with a semi-column:

```
y = [1; 2; 3];
```

Creating Matrices and Strings

Suppose we now have a matrix $A \in \mathbb{R}^{3\times 3}$:

$$A = \begin{bmatrix} 1 & 3 & 2 \\ 1 & 1 & 2 \\ 3 & 1 & 4 \end{bmatrix}$$

We can use the commands in the previous slide to create it, as follows:

```
A = [1 3 2; 1 1 2; 3 1 4];
```

A string is obtained by including the text in single quotes:

```
z = 'string';
```

Sometimes, when importing your data, you could be dealing with strings and, in the next slides, you can find some useful commands to process them.

Useful Commands for Dealing with Strings

Let us suppose that you imported some integers and saved them in string called `S`:

```
S = ['1 2 3 4'];
```

You can use the function `str2num()` to convert `S` into a numeric array `X`:

```
X = str2num(S);
```

In case we have a single number as in the string `S = ['2']`, we can use:

```
x = str2double(S);
```

in order to convert our string into a scalar `x`. In case you are dealing with complex numbers, please remember that the spacing has an impact and it influences the final result:

`x = str2num('2+3i');` is different from `x = str2num('2 +3i');`

Transpose of a Matrix

The nonconjugate transpose of a vector (or matrix) can be obtained by adding a dot and single quote (.') after the variable, or by using the function `transpose()`. Starting from the matrix `A` we defined before, if we write:

```
A_transpose = A.';  or  A_transpose = transpose(A);
```

we can verify that we obtain the transpose of matrix `A`:

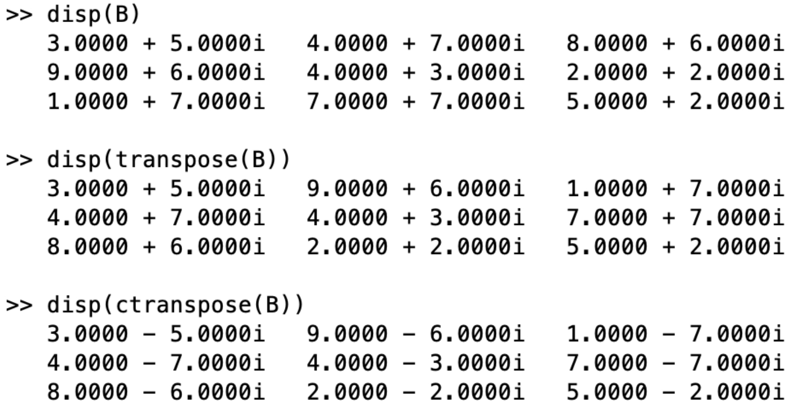
$$\begin{bmatrix} 1 & 1 & 3 \\ 3 & 1 & 1 \\ 2 & 2 & 4 \end{bmatrix} = A^{\mathsf{T}}$$

The conjugate transpose of a vector (or matrix) can be obtained by adding a single quote (') after the variable, or by using the function `ctranspose()`, as follows:

```
A_conjTranspose = A';  or  A_conjTranspose = ctranspose(A);
```

Transpose of a Matrix

⋆ **Linear Algebra Reminder:** Please notice that, for a real matrix like `A`, there is no difference between the conjugate and nonconjugate transpose. However, if we consider a complex matrix $B \in \mathbb{C}^{3\times 3}$, we obtain different results (the function `disp()` prints on screen a given variable):



```
>> disp(B)
   3.0000 + 5.0000i   4.0000 + 7.0000i   8.0000 + 6.0000i
   9.0000 + 6.0000i   4.0000 + 3.0000i   2.0000 + 2.0000i
   1.0000 + 7.0000i   7.0000 + 7.0000i   5.0000 + 2.0000i

>> disp(transpose(B))
   3.0000 + 5.0000i   9.0000 + 6.0000i   1.0000 + 7.0000i
   4.0000 + 7.0000i   4.0000 + 3.0000i   7.0000 + 7.0000i
   8.0000 + 6.0000i   2.0000 + 2.0000i   5.0000 + 2.0000i

>> disp(ctranspose(B))
   3.0000 - 5.0000i   9.0000 - 6.0000i   1.0000 - 7.0000i
   4.0000 - 7.0000i   4.0000 - 3.0000i   7.0000 - 7.0000i
   8.0000 - 6.0000i   2.0000 - 2.0000i   5.0000 - 2.0000i
```

Inverse of a Matrix and System of Equations

The inverse of a matrix can be computed with the following command:

```
A_inverse = inv(A);
```

However, when solving a system of linear equations, it is better to use the *backslash operator*. In other words, the solution to the system $Ax = b$ can be computed either as:

```
x = inv(A)*b;
```
or
```
x = A\b;
```

The former method is slower and less accurate than the latter: this shortcoming of the `inv()` function is particularly evident when the matrix considered has a high condition number.

$\star$ **Linear Algebra Reminder:** The condition number of a matrix gives an indication of the sensitivity of the system to perturbations, and corresponds to the ratio between the largest and smallest singular values. In MATLAB it can be computed by using either `cond()` or `rcond()`, with the latter indicating the reciprocal of the condition number – ranging from 0 (badly conditioned) to 1 (well conditioned).

Matrix Product and Element-wise Matrix Operations

The matrix product between two matrices can be computed in the following two ways:

`AB = mtimes(A,B);` or `AB = A*B;`

In practice, we normally use ‘`*`’ to perform matrix multiplication. It is important to notice that, if we put a dot in front of the symbols used for the standard operations, they will be performed element-wise. Multiplication, division and matrix power (which in the standard case can be computed as `^` or `mpower(A,B)`) will be, instead, computed element by element as:

```
A.*B;
A./B;
A.^B;
```

Selecting Elements Inside a Matrix

While with the square brackets we can create matrices and vectors, with the regular brackets we can extract the elements inside. Let us consider again a matrix A defined as:

```
A = [1 3 2; 1 1 2; 3 1 4];
```

We can select the element in position (2,2) and change its value to 5 as follows:

```
A(2,2) = 5;
```

Please remember that the first argument refers to the row index, while the second to the column index. If we want instead to select the last element of the first row, we can write:

```
A(1,end);
```

In general, the value of `end` corresponds to the index of the last element in the row/column.

The Colon (:) Operator and `linspace()`

The colon operator has a wide variety of uses in MATLAB. In general, it can be used to create a vector if we give it the first and last element. For example, we can obtain a vector with all the integers from 1 to 10 by writing:

```
x = 1:10;
```

We can also specify the distance between one element and the next one, as follows:

```
y = 1:0.01:10;
```

Here `y` will contain all the numbers from 1 to 10, in intervals of 0.1. A similar result can be obtained by using the function `linspace()`, which takes as third argument the number of points we want to consider:

```
z = linspace(1,10,1000);
```

Vector `z` will contain 1000 equally spaced elements between 1 and 10.

Indexing with the Colon (:) Operator

An important use of the column operator consists in the indexing of matrices. Let us consider the matrix A defined before and the following command:

```
A(1,:);
```

The command above selects the first row of A, while we can select the first column by writing:

```
A(:,1);
```

We can also select sub-matrices, like the 2×2 matrix in the lower right part of A:

```
A(2:end,2:end);
```

Please notice how we used the `end` operator to quickly access the last element, without necessary remembering that A is a 3×3 matrix. The column operator can also be used to vectorize a matrix: `A(:)` will return a column vector of length 9 with the elements of A concatenated vertically column by column.

Linear Algebra Recap: Vector Norms

Given a vector $\mathbf{x} \in \mathbb{R}^n$, we can define the following norms:

$$\|\mathbf{x}\|_1 = \sum_{i=1}^{n} |x_i|,$$

$$\|\mathbf{x}\|_2 = \sqrt{\mathbf{x}^T\mathbf{x}} = \sqrt{\sum_{i=1}^{n} |x_i|^2},$$

$$\|\mathbf{x}\|_\infty = \max_{i=1,\dots,n} |x_i|,$$

where x_i indicates the *i*-th component of vector $\mathbf{x}$.

Università della Svizzera italiana | Institute of Computing CI

Linear Algebra Recap: Matrix Norms

Given a matrix $A \in \mathbb{R}^{n\times m}$, with a_{ij} representing the element of the matrix in position (i,j), we can define the following norms:

$$\|A\|_1 = \max_{j\in 1,\dots,m} \sum_{i=1}^{n} |a_{ij}|,$$
$$\|A\|_2 = \sqrt{\lambda_{\mathsf{max}}(A^*A)},$$
$$\|A\|_\infty = \max_{i\in 1,\dots,n} \sum_{j=1}^{m} |a_{ij}|,$$

where $\lambda_{\mathsf{max}}(A^*A)$ is the largest eigenvalue of matrix A^*A, and A^* the conjugate transpose.

Computing Norms and Eigenvalues

We can play around with the formulas in the two recap slides and compute analytically the different vector and matrix norms. The only commands you are probably missing in your toolbox are `max()` to compute the maximum, `abs()` for the absolute value and `eigs()` to compute the eigenvalues of the matrix. Please remember that:

```
[∼,D] = eigs(A);
```

will return a diagonal matrix with the 6 largest eigenvalues of `A` (while the tilde suppresses the first output of the function). If we want all the eigenvectors and eigenvalues, we write:

```
[V,D] = eig(A);
```

and save in the diagonal matrix `D` the eigenvalues and in `V` the associated eigenvectors. We can then compare the results obtained with the norms computed by using `norm()`, which accepts as second argument the norm type (e.g., `norm(A,'inf')` returns the infinity norm).

Some Specific Matrices

A $n \times m$ **identity matrix** can be generated with the command:

```
A = eye(n,m);
```

while a **matrix of zeros** can be intuitively obtained as:

```
A = zeros(n,m);
```

MATLAB supports also sparse structures, so we can create a **sparse matrix** as:

```
A = sparse(n,m);
```

$\star$ **Friendly advice:** Depending on your problem, using sparse matrices can be highly beneficial and can help you saving a lot of memory, but always remember to double check if the function you are planning to use supports sparse matrices.

Some Specific Matrices

A $n \times m$ **matrix of ones** can be generated with the command:

```
A = ones(n,m);
```

while a **random matrix** can be intuitively obtained as:

```
A = rand(n,m);
```

In case we are interested in having only **random integers**, we can write:

```
A = randi(upper_limit,n,m);
```

where `upper_limit` indicates up to which integer we can extract the random numbers. If we want, instead, to extract the **diagonal** of a matrix and save it in a vector, we can write:

```
x = diag(A);
```

Concatenation and Rotation of Matrices

The same syntax we used to create vectors and matrices can be used to concatenate matrices:

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

for example, concatenates on the first row the 25×25 matrix `A` and the 25×75 matrix `B`, and then concatenates this result vertically with the 15×100 matrix `C`.

$\star$ **Friendly advice:** Always double check the dimensions of the matrices you are concatenating if you want to avoid nasty bugs in your code. Use the command `[m,n] = size(A)` to return the number of rows and columns of a matrix in case you have any doubt.

Another useful command is represented by the following function:

```
A_rot = rot90(A,2);
```

which allows us to rotate by multiples of 90 degrees the entries of a matrix (180 degrees in this case). This command can be relevant in case we are dealing, e.g., with images.

Logical Operators

The logical operators return, as usual, the value 0 or 1, depending on whether or not a specific condition is verified. Some examples are: <, >, <=, >=, == (equal) and ∼= (not equal). An important fact is that logical operators can be applied to matrices; so, for example:

```
B = A > 0.5;
```

creates a logical matrix B, which contains in every position 0 or 1 depending on whether the condition of the entry being bigger than 0.5 is satisfied. Instead the command:

```
c = sum(sum(A > 0.5));
```

returns a variable c, which counts all the elements of A bigger than 0.5. We introduced also the function `sum()`, which produces a row vector containing the sum of every column of a given matrix (if instead the input is a vector, the function will just sum all the elements).

Conditional Statements: If/else

The general structure of the if/else construct is the following:

```
if b == -1
    disp('-1');
elseif b == 0
    disp('0');
elseif b == 1
    disp('1');
else
    disp('other');
end
```

where we used the function `disp()` to display the result as a string.

Università della Svizzera italiana | Institute of Computing CI

Conditional Statements: Switch/case

The general structure of the switch/case construct is the following:

```
switch b
    case -1
        disp('-1');
    case 0
        disp('0');
    case 1
        disp('1');
    otherwise
        disp('other');
end
```

Please notice the `end` which closes the statement.

Università della Svizzera italiana | Institute of Computing CI

Conditional Loops: While and For

The general structure for a conditional loop with a while is:

```
while [condition]
    [body];
end
```

while a double for loop can be written as:

```
for [condition]
    for [condition]
        [body];
    end
end
```

Please notice that the for loop accepts as `[condition]` also a vector.

A Note on Loops and Performance

⋆ **Friendly advice:** Always, always, always try to vectorize your computations whenever possible! A smartly vectorized code runs significantly faster in MATLAB, so try to avoid for loops unless they are necessary (or the scale of the application does not justify the effort).

Scripts and Functions

You are free to organize your code in the way you prefer, but a tidy way to do it consists in creating a `main.m` script in which you include the core of your computations and several functions files (maybe included in a ‘`functions/`’ folder) each containing a single function. The general definition of a function in MATLAB is the following:

```
function [output1, output2, ...] = functionName (input1, input2, ...)
    [body];
end
```

and you normally save it in a file named `functionName.m`. You can then easily call it inside your `main.m` script by running the command:

```
[output1, output2, ...] = functionName(input1, input2, ...);
```

Including the commands `close all`, `clear all` and `clc` at the beginning of your `main.m` script clears the console and the workspace, and closes all the additional objects (e.g., figures).

On the Principles of Plotting

Producing good plots is an art in itself and the visualization of your results covers a significant role in all those situations in which you have to communicate with other people (and this includes your final project report and presentation!). Let us start with an example of figure and analyze the different elements:

```
figure()
plot(trueGamma,'LineWidth',1.5)
hold on
plot(approxGamma,'LineWidth',1.5)
title('True Gamma vs Approximation')
xlabel('T')
ylabel('Gamma')
legend('true gamma','approx gamma')
axis([0 T 0.9 1.1])
```

On the Principles of Plotting

The command `figure()` creates a figure handle in which your plot will be drawn. If you want to close the figure, you can use the command `close` (or `close all` for multiple figures).

The command `hold on` allows you to plot two different lines on the same plot. All other commands are rather intuitive, except maybe `axis`, which allows you to specify the upper and lower limit for the x and y axis, by passing as input to the function a vector with 4 components.

Apart from the function `plot()`, which is widely used to plot a 2D lines, there are several other plot types. Among those, it is worth mentioning `semilogx()`, `semilogy()` and `loglog()`, which allow you to plot your data in logarithmic scale on the x axis, on the y axis or on both, respectively. The function `surf()` produces instead a 3D coloured surface, while `plot3()` plots lines and points in a 3D space.

⋆ **Friendly advice:** MATLAB is a powerful visualization tool, so explore and experiment!

Additional Material and References

This set of slides was aimed at introducing MATLAB and some of its functionalities, along with some practical advices and insights. However, it cannot cover every possible topic and, for this reason, you should check also other references, *in primis*:

`https://www.mathworks.com/learn/tutorials/matlab-onramp.html`

which consists in 2 hour interactive tutorial about all the basic functionalities of MATLAB. In case you do not already have a MathWorks account, you can create a new one by using your USI credentials. Furthermore, at the link:

`https://matlabacademy.mathworks.com/`

you can visualize all the additional application-specific courses you have free access to.

⋆ Exercise 1: Plots and Functions

Consider the following function of x in the interval $[-2\pi, 2\pi]$:

$$f(x) = e^{-\frac{x^2}{5}} \sin(2x).$$

Create a script `exercise1.m` which solves the following tasks:

1. Plot the function for 200 linearly spaced points in the given interval using a for loop.
2. Define a function handle for $f(x)$ and plot the function for 1000 linearly spaced points in the given interval **without** using any for loop.
3. Add title and axis labels to the two plots and save them in EPS format.

⋆ Exercise 2: Taylor Series

The value of $\cos(x)$ can be approximated by using the following Taylor series:

$$\cos(x) = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n}}{(2n)!}.$$

Of course, for the purpose of implementation, we are interested in a truncated version of this series and we will indicate with N the number of terms considered.

Solve the following tasks:

1. Write a function `approxCos()` which approximates $\cos(x)$ according to the formula above. The function takes as inputs the number of points N and the point x in which we want to compute the approximation, and returns the approximated value of the function. Save this function in a folder called '`functions/`'.
2. Write a script `exercise2.m` in which you call `approxCos()` to compute the approximate value of $\cos(x)$ for $x = 0.5$ and $x = 35$. Compute the L1-norm of the difference between the approximation and the exact solution in both cases and plot it in logarithmic scale (use `semilogy`) for $N = 1, \dots, 20$ in the same figure (use `hold on`).

⋆ Exercise 3: Coin Toss

We toss a fair coin 10 times in a row.

Solve the following tasks:

1. What is the probability of obtaining exactly 5 heads in a row (no more, no less)? Use the concepts explained in tutorial 1 to compute this probability.
2. Write a script `exercise3.m` in which you verify empirically your results by running 1000 simulations of this experiment.

⋆ Exercise 4: Stop and Plot

We toss a fair coin and we stop when we obtain three heads or tails in a row.

Solve the following tasks:

1. Write a function `coinToss()` that simulates the experiment described above and returns after how many tosses we stopped.
2. Write a script `exercise4.m` in which you call your function 100, 1000 and 10000 times and save the results of the simulations in three vectors `v1`, `v2` and `v3`. Plot these vectors in three different histograms.

⋆ Exercise 5: Independent Subexperiments

We conduct a series of independent sub-experiments in which only two outcomes are possible: ‘success’ or ‘failure’, each one with the same probability.

Solve the following tasks:

1. Compute the probability of obtaining 3 successes in 5 trials.
2. Verify your results in a script `exercise5.m` by running 10000 simulations.