Numerical recipes for environmental sciences with MATLAB

Fabio Durastante

Mar 31, 2022

CONTENTS

1 An introduction to MATLAB **3**
- 1.1 The MATLAB Desktop . . . 3
- 1.2 Arithmetic operations and order of operations . . . 4
- 1.3 Variables . . . 4
- 1.4 Some functions you should remember . . . 6
- 1.5 Script creation . . . 7
- 1.6 Creating Functions . . . 8
- 1.7 Arrays and Matrices . . . 9
- 1.8 Selection and Conditional Structures . . . 16
- 1.9 Cycles and Nested Cycles . . . 18
- 1.10 Some exercises . . . 18

2 Reading, writing and plotting data **23**
- 2.1 Writing data to screen and to file . . . 32

3 Modeling evolutionary problems **33**
- 3.1 Ordinary differential equations . . . 33
- 3.2 A completely worked out example . . . 34
- 3.3 Chemical kinetics . . . 39
- 3.4 A simple epidemiological model . . . 40

4 Fitting data to models **43**
- 4.1 Least square approach . . . 43
- 4.2 Fitting data to a differential model . . . 47

5 Graphs and Networks **53**
- 5.1 A social network made of dolphins . . . 54
- 5.2 Finding communities . . . 60
- 5.3 Centrality . . . 65

6 Working with images **69**
- 6.1 Denoising images . . . 75
- 6.2 Deblurring images . . . 80

7 Linear Algebra **87**

Bibliography **89**

Proof Index **91**

Numerical simulations are computations we run on a computer with programs implementing a mathematical model for a chemical, physical or biological system. We need them to study the behavior of processes whose mathematical formulations are too complex to provide analytical solutions. The computational science area is itself a rapidly growing field. While the largest and most accurate simulations often use advanced computing capabilities, there is an ample layer of small and intermediate problems across many disciplines that we can face with easier-to-handle tools. This course will address one of such tools called MATLAB. We will use it to perform small-scale computer simulations. In the first place, we are going to introduce the *programming language* on its own and take some familiarity with it. Then, we will apply it to solve some problems in Earth, Life, and Chemical sciences.

Scheduling: Two lessons per week of two hours each from the 1st of March. Tuesday and Thursday from 9.00 to 11.00

Modalities: The course will take place online in synchronous mode.

CHAPTER

ONE

AN INTRODUCTION TO MATLAB

MATLAB / Simulink is a software tool for:

- Performing mathematical calculations and signal processing,
- Data analysis and visualization: you have several graphical tools,
- Modeling of physical systems and phenomena,
- Testing and simulation of engineering projects.

1.1 The MATLAB Desktop

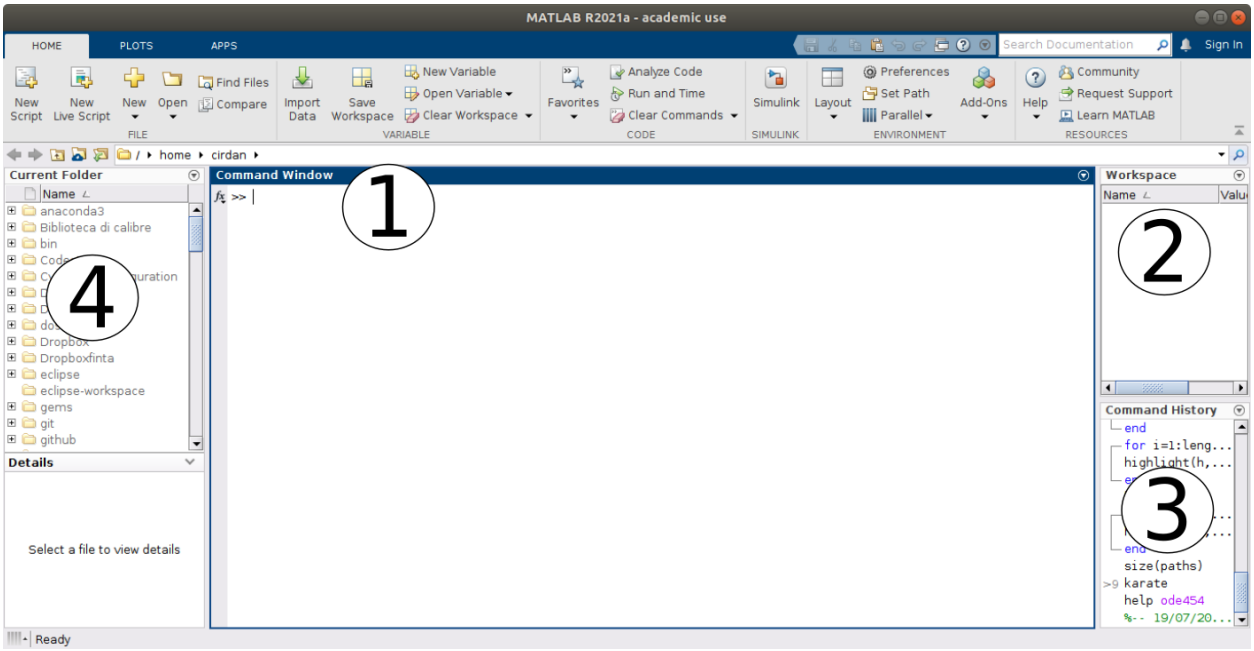


Fig. 1.1: The MATLAB Desktop

1. The **command window** is where you type MATLAB commands following the prompt:

```
>>
```

1. The **workspace** window shows all the variables you have defined in the current session. Variables can actually be manipulated within the workspace window.

2. The **command history** shows all MATLAB commands you have used recently, including past sessions as well.
3. The **current folder** shows all files in whichever folder has been selected to be the current folder.

1.2 Arithmetic operations and order of operations

- The basic operations are addition (+), subtraction (−), the multiplication (`*`), division (`\`), exponentiation (`^`),
- The order of the operations is the *canonical* one used in mathematics and follows the usual conventions of a scientific calculator
    1. Complete all operations in parentheses `()` using the following rules of precedence,
    2. Exponentiation (left to right)
    3. Multiplication and division (left to right)
    4. Addition and subtraction (from left to right)

Example

For example, consider the following operations:

```
30/5*3
5*2^4+4*(3)
-1^8
8^(1/3)
```

1.3 Variables

As in any programming language, MATLAB makes use of **variables**, which, in computer science, are data containers located in a portion of the memory and intended to contain values, which can (in general) be modified during the running a program.

A variable is characterized by a name (usually intended as a sequence of characters and digits) which must follow a set of *conventions* that depends on the language being used. In MATLAB the following conventions must be used:

1. Variable names must start with a letter,
2. Names can include any combination of letters, numbers, and *underscores*,
3. The maximum length for a variable name is 63 characters,
4. MATLAB is **case sensitive**. The variable named `bAll` is different from the variable named `ball`,
5. It is a good idea to avoid the following names: `i`, `j`, `pi`, and more generally all predefined MATLAB function names such as `length`, `char`, `size`, `plot`, `break`, `cos`, `log`, etc.
6. It is good practice to call variables with intelligible names, that is, with names that reflect their use within the program rather than using gods generic variable names such as `x`, `y`, `z`.

Tip: If we want to calculate the surface air of a sphere $A = 4\pi r^2$ of radius $r = 5$, it is better to write

```
radius = 5;
surface_area = 4 * pi * radius ^ 2
```

in the place of

```
r = 5;
A = 4 * pi * r ^ 2
```

When we reopen the second code in a month, the chances of remembering what we meant will be pretty scarce (or, in my case, even if I open it in two hours).

From the example we have just entered in the **workspace** we observe several things

```
radius = 5;
surface_area = 4*pi*radius^2
```

1. a `radius` variable of type `double` has been created,
2. a memory location for the variable `radius` has been allocated and initialized to the value 5,
3. the `;` at the end of the instruction suppresses the screen printing of the content of the variable,
4. the variable `surface_area` is instead created and allocated using instead one of MATLAB's predetermined quantities, the value of π, and the content of the variable `radius` we defined earlier. Since we didn't finish the second statement with a `;`, we see its value printed in the **command window**

```
surface_area =

  314.1593
```

At this point we can change the value of the `radius` variable simply by reassigning it to a new amount *without* altering the value stored in the **workspace** of the `surface_area` variable.

To display the contents of a variable, you can use the `disp` command, for example:

```
disp ("The surface area of the sphere is:"); disp (area_surface);
```

which will produce in the **command window**:

```
The surface area of the sphere is:
   314.1593
```

1.3.1 Character variables and strings

We are not obliged to use only numeric variables. For **example** we can write in the **command window**:

```
student='C.F. Gauss'
```

that will print us

```
student =

    'C.F. Gauss'
```

We have built an **array** of `char`, that is a variable named `student` to which we have assigned a memory space and in which we have inserted the characters `C.F. Gauss`. If in doubt, we can query MATLAB about the type of the variable with the command

```
whos student
```

that will give us back

```
  Name          Size             Bytes  Class    Attributes

  student       1x10                20  char
```

A variation on the **array** of `char` is to define a `string` object instead, i.e. assign

```
student_string = "C.F. Gauss"
```

so `whos student_string` will tell us instead

```
Name                  Size             Bytes  Class     Attributes

student_string       1x1                156  string
```

Summing up:

- An *array of* `char` is a sequence of characters, just as a numeric vector is a sequence of numbers. Its typical use is to store short pieces of text as character vectors,
- A `string` object, that is, an *array* of `string` is a container for parts of text. *Arrays of Strings* provide a variety of functions for working with text as data.

1.4 Some functions you should remember

MATLAB contains a large number of already implemented mathematical functions, some of them, the most frequently used ones, are listed in Table 1.1

Table 1.1: alcune funzioni di base.

Function	MATLAB	Function	MATLAB
Cosine	`cos`	Square root	`sqrt`
Sine	`sin`	Exponential	`exp`
Tangent	`tan`	Logarithm (base 10)	`log10`
Cotangent	`cot`	Logarithm (natural)	`log`
Arccosine	`acos`	Round to the nearest integer	`round`
Arctangent	`atan`	Round up to the integer $\leq$	`floor`
Arc tangent	`acot`	Round up to the integer $\geq$	`ceil`

Danger: The trigonometric functions thus expressed take the inputs in radians, that is, the inverse functions return the angle in radians. Versions that use degrees are invoked with the suffix `d`, e.g., `cosd`, `acosd`.

If you encounter a function you don't know how to use, you can query the MATLAB help from the **command window** with, for example,

```
help cosd
```

which will print essential information

```
cosd   Cosine of argument in degrees.
   cosd(X) is the cosine of the elements of X, expressed in degrees.
```

(continues on next page)

(continued from previous page)

```
   For odd integers n, cosd(n*90) is exactly zero, whereas cos(n*pi/2)
   reflects the accuracy of the floating point value for pi.

   Class support for input X:
      float: double, single

   See also acosd, cos.

   Documentation for cosd
   Other functions named cosd
```

so you can then access the complete information using the link *Documentation for ….*

A final couple of extremely useful functions are the

- `clear` functions which clear the **workspace** of all variables, i.e. `clear variablename1 variablename2` which only clears the variables `variablename1` and `variablename2`,
- `clc` which clears the content printed in the **command window**.

1.5 Script creation

All (or almost) the commands seen so far are actually **scripts** or **functions** pre-built and made available in the general environment. MATLAB allows the user to build his own scripts and functions for solving specific problems.

A **script file** is simply a *collection* of MATLAB executable commands and, in some more refined cases, interfaces to external software produced in C or Fortran.

To create a new script just *click* on the **New script** icon, Fig. 1.2, which will open a new *editor* window where you can write your own program.

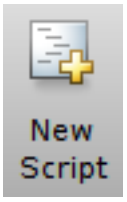
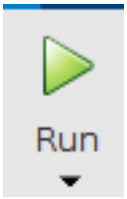


Fig. 1.2: Create a new script.

As we said, a script is nothing more than a sequence of commands to be inserted in the editor window. In order to *execute* the script it must be saved in the **Current Folder**.

Warning: The naming conventions for scripts are the same as for variable names ({ref} `sec-variables`), in particular it is extremely important to avoid using predefined MATLAB function names.

You can **run** the script

1. Click on the **Run** button 

2. Enter the name with which the script was saved in the **command window** (without the `.m` extension).

Esempio

Let's turn our code for calculating the surface area of a sphere into a script. That is, we write in the editor:

```
radius = 5;
surface_area = 4 * pi * radius ^ 2;
disp ("The Surface Area of the Sphere is:"); disp (surface_area);
```

We save the file as `spherearea.m` and run it once with the **Run** button and once by writing `spherearea` in the **Command Window**.

To summarize Script files are extremely useful when you intend to run sequences of many MATLAB commands. For example, let's imagine a sequence of calculations divided into n commands, after having entered them all in the *prompt* we realize that the value assigned to a variable in the first command is wrong, or we want to change it. Working only in the **command window** we should fix the error in the first command, then rerun the other $n - 1$ commands.

If we had produced a script instead, we could simply correct the first command and rerun the entire sequence by pressing a single key or typing a single command: the name of the script.

1.6 Creating Functions

A function (also called, depending on the programming language, routine, subroutine, procedure, method), is a particular syntactic construct that groups within a single program, a sequence of instructions in a single block. Its purpose is to carry out an operation, action, or processing in such a way that, starting from certain *inputs*, it returns certain *outputs*.

In MATLAB a function named `myfunction` is declared as`function [y1, ..., yN] = myfunction (x1, ..., xM)` which accepts as *input* `x1, ..., xM` and returns as * output * `y1, ..., yN`. This declaration **must** be the first executable statement of the file that contains the function. Valid function names begin with an alphabetic character and can contain letters, numbers, or *underscores*.

You can save a function in

- a function file that contains only function definitions. The file name *must* exactly match the name of the first function in the file;
- a script that contains commands and function definitions. In this case the functions *must* be contained at the end of the file and the *script* cannot have the same name as any of the functions contained within it.

Files can include multiple local or nested functions. In order to produce human readable code, it is recommended to always use the `end` keyword to indicate the end of any function within a file.

In particular, the `end` keyword is required whenever:

- any function in a file contains a nested function,
- the function is a *local* function within a file that contains only functions and, in turn, each local function uses the keyword `end`,

- the function is a *local* function inside a *script*.

To interrupt the execution of a function before reaching the final `end`, you can use the `return` keyword which, as the name suggests, returns control to the script that invoked the function. A direct call to the script or function that contains `return` does not invoke any source program and instead returns control to the command prompt.

Example

We transform our code for calculating the surface area of a sphere into a function. That is, we write in the editor: `` matlab function [surface_area] = areasphere(radius)

surface_area = 4 * pi * radius ^ 2; end `` We save the file as `areasfera.m` and we can execute the function directly in the **command window** such as

1. `area_surface = areasphere (radius),`
2. `area_surface = areasphere (radius);,`
3. `areasphere (radius)`. Note the differences.

1.7 Arrays and Matrices

We have been discussing the use of scalar variables so far, however in MATLAB, data is naturally represented as **matrices**, **arrays** and, more generally, **tensors**. You can apply all linear algebra operations to arrays, plus you can create common grids, combine existing arrays, manipulate the shape and content of an array, and use different indexing modes to access their elements.

In fact, we have already surreptitiously worked with arrays, since in MATLAB even scalar variables are nothing more than one-dimensional arrays, in mathematical notation an $a \in \mathbb{R}$ is always a $a \in \mathbb{R}^{1\times1}$. In fact, if we write in the **command window**:

```
a = 100;
whos a
```

the system will give us back

```
Name        Size              Bytes  Class     Attributes

a           1x1                   8  double
```

Now let's imagine that we have a specific set of data, which we want to arrange in a matrix. We can do this using the semantics of square brackets `[]`.

A single line of data contains spaces or commas `,` between elements and uses a semicolon `;` to separate rows.

For example, if we want to create a single row of three numeric elements

```
v = [ 1 2 3]
```

or

```
v = [ 1, 2, 3]
```

The resulting matrix size is 1 by 3, as it has one row and three columns. A matrix of this form is called a **row vector** and if we call the `isrow (v)` function we get the answer

```
ans =

  logical

   1
```

Similarly, we can construct a **column vector** as

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

so we have that `iscolumn(w)` will return us instead

```
ans =

  logical

   1
```

More generally, we can construct an array of 4×4 elements like

```
A = [ 1 2 3 4; 5 6 7 8; 9 10 11 12; 13 14 15 16]
```

or as

```
A = [ 1, 2, 3, 4; 5, 6, 7, 8; 9, 10, 11, 12; 13, 14, 15, 16]
```

or

```
A = [ 1, 2, 3, 4
      5, 6, 7, 8
      9, 10, 11, 12
      13, 14, 15, 16]
```

In all cases we will see ourselves returned

```
A =

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

We can investigate the dimensions of an array using the `size (A)` function, which in the previous case will return us

```
ans =

     4     4
```

1.7.1 Default constructors

Constructing arrays simply by inserting values into them sequentially or explicitly is clearly cumbersome (and very boring). For this purpose MATLAB has the constructors listed in Table 1.2.

Table 1.2: Constructors for arrays and matrices.

Function	Result
`zeros`	**Create an array of all zeros** To build the matrix $0 = (O)_{i,j}\ i, j = 1, \dots, n$ we write `O = zeros(n,m);`
`ones`	**Create an array of all ones** To build the matrix $1 = (E)_{i,j}\ i, j = 1, \dots, n$ we write `E = ones(n,m);`

`rand`	**Create an array of evenly distributed random numbers** in $[0, 1]$.
	`R = rand(n,m);`
	We can generate evenly distributed numbers in the range $[a, b]$ as
	`R = a + (b-a).*rand(n,m);`

`eye`	**Identity matrix** builds the identity matrix $(I)_{i,i} = 1$, $i = 1, \dots, n$ and 0 otherwise `I = eye(n)`.
`diag`	Creates a **diagonal matrix** or extracts the diagonal elements of a given matrix. If `v` is a vector of length n, then `diag(v)` is a matrix whose leading diagonal is given by the elements of `v`. If `A` is a matrix then`diag(v)` is a vector containing the elements of the leading diagonal of A.

Other functions of the same type are `true`, `false`, `blkdiag` and theirs operation can be explored using the `help` function.

The second set of functions extremely useful for building matrices is the one that deals with managing concatenations. These can be obtained either by using the notation with square brackets `[]`, for example,

```
A = rand(5,5);
B = rand(5,10);
C = [A,B];
```

builds from matrices $A \in\ mathbbR^{5\times5}$, $B \in \mathbb{R}^{5\times10}$, the matrix $C \in \mathbb{R}^{5\times15}$ obtained by placing next to each other all the columns of A followed by those of B. Similarly, vertical concatenation can be achieved as

```
A = rand(8,8);
B = rand(12,8);
C = [A;B];
```

which will generate the $C \in \mathbb{R}^{20,8}$ matrix obtained by stacking all lines of A followed by lines of B.

Danger: The concatenation operations must be done between matrices of compatible size, there can be no "leftovers" between the dimensions in question. To try to get an error, execute:

```
A = rand(5,5);
B = rand(5,10);
C = [A;B];
```

which will return

```
Error using vertcat
Dimensions of arrays being concatenated are not consistent.
```

Instead of the notation with square brackets it is possible to make use of the functions `horzcat` and `vertcat` which correspond, respectively, to the concatenations of the form `[,]` and `[;]`.

If we want to build instead a diagonal block matrix starting from the diagonal blocks we can use the `blkdiag` function whose `help` gives us exactly

```
blkdiag  Block diagonal concatenation of matrix input arguments.

                                    |A 0 .. 0|
   Y = blkdiag(A,B,...)  produces   |0 B .. 0|
                                    |0 0 ..  |

   Class support for inputs:
      float: double, single
      integer: uint8, int8, uint16, int16, uint32, int32, uint64, int64
      char, logical
```

1.7.2 Slicing: access elements

The most common way to access a particular element of an array or matrix is to explicitly specify the indices of the elements. For **example**, to access a single element of an array, specify the row number followed by the column number of the element:

```
A = [ 1 2 3 4
      17 8 2 1];
A(2,1)
```

which will print in the **command window** `17`, i.e. the element in row 2 and column 1 position of `A` ($a_{2,1}$).

We can also refer to multiple elements at a time by specifying their indices in a vector. For **example**, we access the first and fourth elements of the second line of the previous `A` by doing

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

which will return

```
ans =

    17     1
```

To access elements in a range of rows or columns, you can use the colon operator `:`. For example, we can access the elements from the first to the fifth row and from the second to the sixth column of a A matrix as

```
A = rand(10,10);
A(1:5,2:6)
```

If you want to scroll to the end of a dimension, you can replace the value on the right in the `:` with the keyword `end`, for example:

```
A(1:5,2:end)
```

Instead, using `:` without start/end values extracts all entries of the relative dimension, for example, 5 column, `A(:, 5)`, or 4th to 7th columns, `A(:,4:7)`.

Note: In general, indexing can be used to access elements of any array in MATLAB **regardless** of data type or size.

The last way to *slice* a vector we want to talk about is by using **logical vectors**. Using the `true` and `false` logical flags is especially effective when working with conditional statements. For **example**, suppose we want to know if the elements of one A array are greater than the corresponding elements of another B array. The comparison operator applied to the two arrays returns a logical array whose elements are 1 when an element in A satisfies the comparison with the corresponding element in B.

```
A = [1 2 6; 4 3 6];
B = [0 3 7; 3 7 5];
ind = A>B
A(ind)
B(ind)
```

gives us back

```
ind =

  2×3 logical array

   1   0   0
   1   0   1


ans =

     1
     4
     6


ans =

     0
     3
     5
```

Comparison operators are not the only ones for which this can be done, MATLAB itself implements a number of useful functions for this purpose, see `isnan`, `isfinite`, `isinf`, `ismissing`. It is also possible to combine several requests together using the **logical operators**, see the section on *Selection and Conditional Structures* below.

1.7.3 Operations between arrays and matrices

MATLAB supports all operations that make sense in terms of linear algebra, therefore matrix-vector products, sums of matrices and vectors, transposition, conjugation, *etc.*, with the same consistency constraints between the operators. Specifically, the product $A\mathbf{v}$ with $A \in \mathbb{R}^{n \times k_1}$ and $\mathbf{v} \in \mathbb{R}^{k_2}$ is possible if and only if $k_1 \equiv k_2$, let's consider for example

```
A = [1 2 3 4;
     4 3 2 1;
     2 4 3 1;
     3 2 4 1];
v1 = [1,2,3,4];
v2 = [1;2;3;4];
```

and let's try to calculate

- `A*v1` from which we get an error:

```
Error using  *
Incorrect dimensions for matrix multiplication. Check that the number of columns in␣
 ↪the first matrix matches the number of rows in the second matrix. To perform␣
 ↪elementwise multiplication, use '.*'.
```

since we are trying to do an operation that does not make sense from the point of view of linear algebra.

- `A*v2` instead it is the correct operation and we get

```
ans =

    30
    20
    23
    23
```

- `v1*A` also this operation makes sense from the matrix point of view and in fact we get

```
ans =

    27    28    32    13
```

- `v2*A` it has the same problem as the first, that is, we again have dimensions that are not consistent

```
Error using  *
Incorrect dimensions for matrix multiplication. Check that the number of columns in␣
 ↪the first matrix matches the number of rows in the second matrix. To perform␣
 ↪elementwise multiplication, use '.*'.
```

- `A*v1'` here we introduce the transpose-conjugate operation (which, since we are using vectors of real numbers, coincides with the simple transposition operation `. "), which leads us to have the correct dimensions and returns

```
ans =

    30
    20
    23
    23
```

- `A*v2'` transposition in this case makes the dimensions of `v2` incompatible so we get the same error again

```
Error using  *
Incorrect dimensions for matrix multiplication. Check that the number of columns in
 ↪the first matrix matches the number of rows in the second matrix. To perform
 ↪elementwise multiplication, use '.*'.
```

Warning: The error message we encountered tells us that the operation does not make sense in terms of the usual operations of linear algebra, however it suggests that what we might have intended to do was an **elementwise** product or "element by element ". This is done by using the . * Operator, that is, with a . prefix in front of the product operator which usually means" execute in an element-by-element fashion ". Let's try the first wrong operation we proposed:

```
A.*v1
```

that gives us back

```
ans =

     1     4     9    16
     4     6     6     4
     2     8     9     4
     3     4    12     4
```

What operation did we perform? Try also the exponentiation operations

```
A^3
A.^3
v1^3
v1.^3
```

and describe the result.

The last two operations we have left to describe are + / −. Again we have to worry about the compatibility of the operations we want to carry out. If we want to be sure we are doing the sum of arrays and matrices we have in mind we have to make sure that the dimensions are compatible.

Suppose we have the row and column vectors

```
v = [1 2 3 4]
w = [1
     2
     3
     4]
```

if we want to add or subtract them and obtain a vector row or column, we must ensure that both have the correct size. That is, we must have, respectively,

```
v + w'
```

or

```
v' + w
```

If we inadvertently add the two vectors as

```
v + w
```

we get instead

```
ans =

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

which is instead a matrix! Not exactly what we expected (can you figure out what operation we got?). Equal caution should be exercised in writing addition/subtraction operations between matrices and vectors. Try running:

```
A = pascal(4);
v = [1 3 2 4];
A + v
A + v'
A + 0.5
v + 1
```

It is legitimate to wonder at this point why these operations are performed without returning errors. Even if from the point of view of pure linear algebra these operations are not immediately sensible, from the implementation point of view they allow to simplify (and speed up) a certain number of operations that would otherwise require several lines of code (whose optimization for performance does not is taken for granted) to be implemented.

1.8 Selection and Conditional Structures

Table 1.3: Conditional Structures.

MATLAB Function	Description
`if, elseif, else`	Execute command if the `if` condition is true
`switch, case, otherwise`	Executes one or more groups of commands depending on whether the `switch` variable has the value indicated in the `case`, otherwise it executes the command or group of commands identified by `otherwise`
`try, catch`	Executes the commands in the `try` group, if they return an error it does not block the execution and passes to the commands collected in the `catch`

Let's consider the following example that simulates the flip of a coin.

```
a = rand();
if a < 0.5
  disp('Head!')
else
  disp('Tail')
end
```

With each new execution, `rand ()` generates a random number in $[0, 1]$ with uniform probability. The `if` command checks if the random number generated is < 0.5 and in this case enters the first code group. Otherwise, we have got a number > 0.5 and we fall into the second group of code. We can also decide to simulate a three-sided die in the following way

```
a = rand();
if a < 1/3
```

(continues on next page)

(continued from previous page)

```
  disp('1')
elseif a >= 1/3 & a < 2/3
  disp('2')
else
  disp('3')
end
```

where we used the `elseif` command to have an additional branch.

Note: Within our `if` checks (and more generally) we can combine the result of several logical operations together. These are collected in Table Table 1.4. Other functions that operate on logical vectors and which you can explore by calling the `help` function are `any` and `all`.

Table 1.4: Logical operations

MATLAB Function	Description
`&`	Logical AND
`~`	Logical NOT
`|`	Logical OR
`xor`	Logical exclusive OR

We also see an example of the `switch` type statement.

```
control = input("Insert an integer between 0 and 3:");
switch control
case 0
 A = pascal(4,4);
 disp(A);
case 1
 A = ones(4,4);
 disp(A);
case 2
 A = eye(4,4);
 disp(A);
case 3
 A = rand(4,4);
 disp(A);
otherwise
 disp ("I don't know what to do with this input!")
end
```

The same code could have been implemented with a series of `if` and `elseif` and an `else`, but this approach is quicker if there is no need to enforce many logical checks.

The last control we want to test is `try`. So you can try the following code.

```
try
 A = rand(5,5);
 b = ones(1,5);
 A*b
catch
 disp ("There is some problem with the size!");
end
```

Since the algebraic operation we requested is not well posed (try to execute it outside the `try` operation) the `try` catches the error and instead of stopping the execution it executes the code in the `catch` clause . You can correct the code in the first block and verify that you will not enter the `catch` in that case.

1.9 Cycles and Nested Cycles

Within any program, you can define sections of code that repeat in a loop.

Table 1.5: Strutture Condizionali.

Funzione di MATLAB	Descrizione
`for`	`for` loop to repeat statements a fixed number of times
`while`	The `while` loop repeats its code as long as the condition is `true`
`break`	Forcibly terminates the execution of a `for` or `while` loop
`continue`	Pass control to the next iteration of a `for` or `while` loop
`pause`	Temporarily pause MATLAB execution.

We use a `for` loop to calculate the sum of the first `n` integers.

```
n = input ("Insert an integer n:");
summation = 0;
for i=1:n
  summation = summation + i;
end
fprintf ("The sum of the integers 1 to% d is% d. \ n", n, summation);
```

This is obviously not the best way to do this, for example we could have calculated the same quantity as `sum (1:10)`. Or, use some mathematical ideas to remind us that $S_n = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$. But it helped our demonstration purpose.

Let's now look at an example of a `while` loop.

```
summation = 0;
while summation < 10
  summation = summation + rand();
end
fprintf ("The final value of the sum is:% f \ n", summation);
```

Since we initialized the `summation` variable to 0, the trigger condition of the `while` loop is `true` and so we start iterating. With each new instance of the loop a new random number is generated and added to the `summation` variable. As soon as the `summation` value exceeds 10, the cycle is interrupted and the message is printed on the screen.

1.10 Some exercises

The exercises collected here are mostly intended to verify that you have absorbed this general information about the MATLAB language, so that we can concentrate on implementing algorithms for the solution of some modeling problem.

Exercise 1

The golden constant φ can be expressed in compact form as

$$\varphi = \frac{1+\sqrt{5}}{2}.$$

Let's assume we don't have an algorithm for extracting square roots, then we can try to approximate the value of ϕ using its continued fraction expansion:

$$\varphi = 1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \ddots}}}}}$$

Write a function with the following prototype:

```
function phi = fractionphi(n)
%% FRACTIONPHI takes as input the number of terms to use
% in the continuous fraction approximation of the golden section e
% returns the approximation.
end
```

- To build the function implementation, use a `for` loop. **Hint** organize the calculation starting from the "lowest level" to the highest one.
- Suppose we know that an exact value of φ with 16 significant digits is `1.6180339887498949`. Change the previous function to a new function with the following prototype

```
function [n,phi] = howmanytermsphi(tol)
%%HOWMANYTERMSPHI given in input a tolerance tol on the distance between
 % the approximation of the constant phi and the real value of this function
 % gives us the number of terms needed and the approximation value

phitrue = 1.6180339887498949;

end
```

  To do this we use a `while` loop and the `abs` function (which implements the absolute value) to measure the **absolute error** between our approximation and the true value.

Exercise 2

Let's practice building a *recursive function* now. A sequence linked to the golden constant φ is the Fibonacci sequence, i.e. the sequence of integers $\{F_n\}_n = \{1, 1, 2, 3, 5, ...\}$ given by

$$F_{n+1} = F_n + F_{n-1}, \text{ se } n \geq 1,\ F_1 = 1,\ F_0 = 1.$$

- Implement a *recursive function* that computes the nth Fibonacci number n using only the `switch` conditional structure, by using the following prototype

```
function f = fibonacci(n)
%FIBONACCI Recursive implementation of the Fibonacci sequence. Takes
% in input the number n and returns the nth Fibonacci number Fn.
end
```

- The function thus constructed has an unfortunate flaw, if we feed it n it gives us the nth number, however if we subsequently ask for the $n+1$th the calculation to obtain it has no memory of what we have done and recalculates all the previous ones anyway. Now let's build a **non-recursive version** of the `fibonacci` function. We can achieve it in several ways, but almost certainly we will need a `for` loop.

```
function f = fibonaccinonrecursive(n)
%FIBONACCINONRECURSIVE Non-recursive implementation of the sequence
% Fibonacci. It takes the number n as input and returns a vector
% which contains all the Fibonacci numbers from F0 to Fn.
end
```

- Let's now take advantage of the second implementation we made of the Fibonacci function to build a different sequence. Let us consider the Viswanath sequence [Vis00] thus defined

$$v_{n+1} = v_n \pm v_{n-1},\ n \geq 1,$$

where v_0 and V_1 are assigned at will and the $\pm$ has the following interpretation: with probability $1/2$ we add, with probability $1/2$ we subtract. One idea to implement this function is to use the `sign` function (see what it is for by doing `help sign`). A prototype for this function is for example

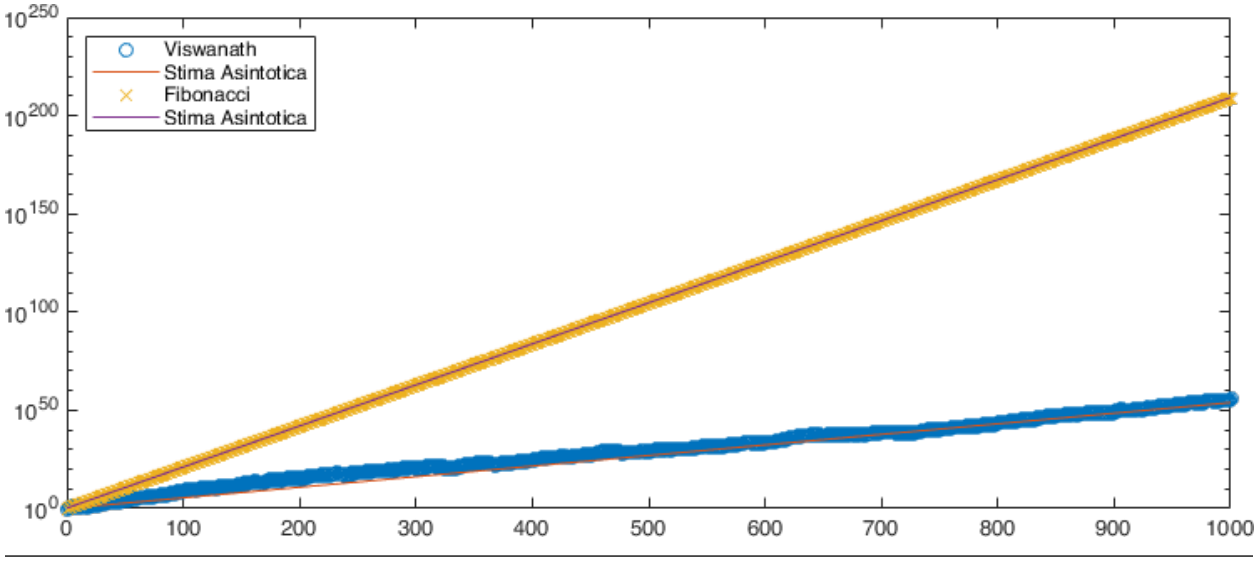
```
function v = viswanath(n,v0,v1)
%VISWANATH Non-recursive implementation of the sequence
% of Viswanath. It takes as input the number n, the values of v0 and v1 and
% returns a vector that contains all Viswanath numbers v0 through vn.
end
```

- Once our function is built, we can visualize what we get with it (and compare it with the Fibonacci sequence) via the script:

```
%% Viswanath sequence test

n = 1000; % Number of terms

% Let's calculate the Fibonacci sequence
f = fibonaccinonrecursive(n);

% We compute the Viswanath sequence
v0 = 1;
v1 = 1;
v = viswanath(n,v0,v1);

% Asymptotic value
c = 1.13198824;
phi = (1+sqrt(5))/2;
figure(1)
semilogy(0:n,abs(v),'o',0:n,c.^(1:n+1),'-',0:n,f,'x',0:n,phi.^(1:n+1),'-');
legend({'Viswanath','Asymptotic estimation','Fibonacci','Asymptotic estimation'},...
    'Location','northwest')
```

In the terminal part of the script `% Asymptotic value` we are going to compare in semi-logarithmic scale on the y axis the absolute values of the numbers $\{v_n\}_n$ and the $\{F_n\}_n$. In particular we can observe that for both sequences we find a number k for which they grow as k^{n+1}. In particular, for the Fibonacci sequence this k is the golden constant φ of the previous exercise, while for the Viswanath sequence it is the value $c = 1.13198824\ldots$ (for a proof see [Vis00]).

Let's take advantage of this also to see how you can get function graphs on MATLAB, to decode the commands in this section use `help`. An example of the graph obtained with the previous script is the following:


Viswanath
Stima Asintotica
Fibonacci
Stima Asintotica
10^250
10^200
10^150
10^100
10^50
10^0
0
100
200
300
400
500
600
700
800
900
1000

CHAPTER

TWO

READING, WRITING AND PLOTTING DATA

One of the transverse usages of MATLAB is as a tool for analyzing and plotting data coming from the most various sources. This material covers the commands and the ideas we may need to perform these tasks.

As you should remember from the introduction to the language, in MATLAB the most natural way of representing data is with matrices, scalars are 1×1 matrices. But how can you **populate these matrices** with the data coming from your experiments? **Are** always **matrices** the **right format**?

We will start with an *example*, from here you can download a **csv** file containing information on the quantity of pm2 particles from pollution stations in the city of London in 2019.

Tip: CSV stands for comma-separated values. These are delimited text files which use a comma to separate values. **Each line** of the file is **a data record**. **Each record** consists of **one or more fields**, separated by commas.

In most of them the separator is indeed a comma, and this justifies the source of the name for the format. Nevertheless, this is not always the case, and other delimiters can be found. In well formatted files each line will have the same number of fields.

Let us work with the data we have just downloaded. First of all, we create a new *MATLAB script* called `londonpollution.m`, and we put the downloaded data in the same folder of the script

```
%% London Pollution Data
% Analysis of the Pollution data from London.

clear; clc; close all;
```

If you look at the first lines of the CSV file you have downloaded, you will see that in the same file appear different types of data, *strings*, *numbers*, *datetime*, and so on. This means that we **cannot store** all these information in a matrix. Matrices only take data of homogeneous type. The right type of variable to use is a `table`.

We use the command `readtable` to load all this information into MATLAB

```
london = readtable('london_combined_2019_all.csv');
```

After these are in memory, you can get some information on the variable by writing in the **command line**:

```
whos london
```

and getting the answer

```
Name            Size                Bytes  Class    Attributes
london      24676x9             16527487  table
```

that tells us that we have loaded a table with 24676 rows, divided in 9 columns. Again from the command window we can look at what are the first rows by doing:

```
head(london)
```

that prints out

```
>> head(london)

ans =

  8×9 table

      city       latitude    longitude    country            utc                        ␣
↪location                parameter     unit      value
    ________    ________    _________    _______    ___________________    ___________
↪___________________    _________    _______    _____

    'London'     51.453     0.070766      'GB'      2019-02-18 23:00:00    'London␣
↪Eltham'                     'pm25'      'ug/m3'      7
    'London'     51.489     -0.44161      'GB'      2019-02-18 23:00:00    'London␣
↪Harlington'                 'pm25'      'ug/m3'      8
    'London'     51.523     -0.15461      'GB'      2019-02-18 23:00:00    'London␣
↪Marylebone Road'            'pm25'      'ug/m3'     17
    'London'     51.521     -0.21349      'GB'      2019-02-18 23:00:00    'London N.␣
↪Kensington'              'pm25'      'ug/m3'      8
    'London'     51.425     -0.34561      'GB'      2019-02-18 23:00:00    'London␣
↪Teddington Bushy Park'      'pm25'      'ug/m3'      8
    'London'     51.495     -0.13193      'GB'      2019-02-18 23:00:00    'London␣
↪Westminster'                'pm25'      'ug/m3'     11
    'London'     51.544     -0.17527      'GB'      2019-02-19 00:00:00    'Camden␣
↪Kerbside'                   'pm25'      'ug/m3'      9
    'London'     51.453     0.070766      'GB'      2019-02-19 00:00:00    'London␣
↪Eltham'                     'pm25'      'ug/m3'      7
```

Let us now try to add some commands to our script to produce plots showing us information on the data.

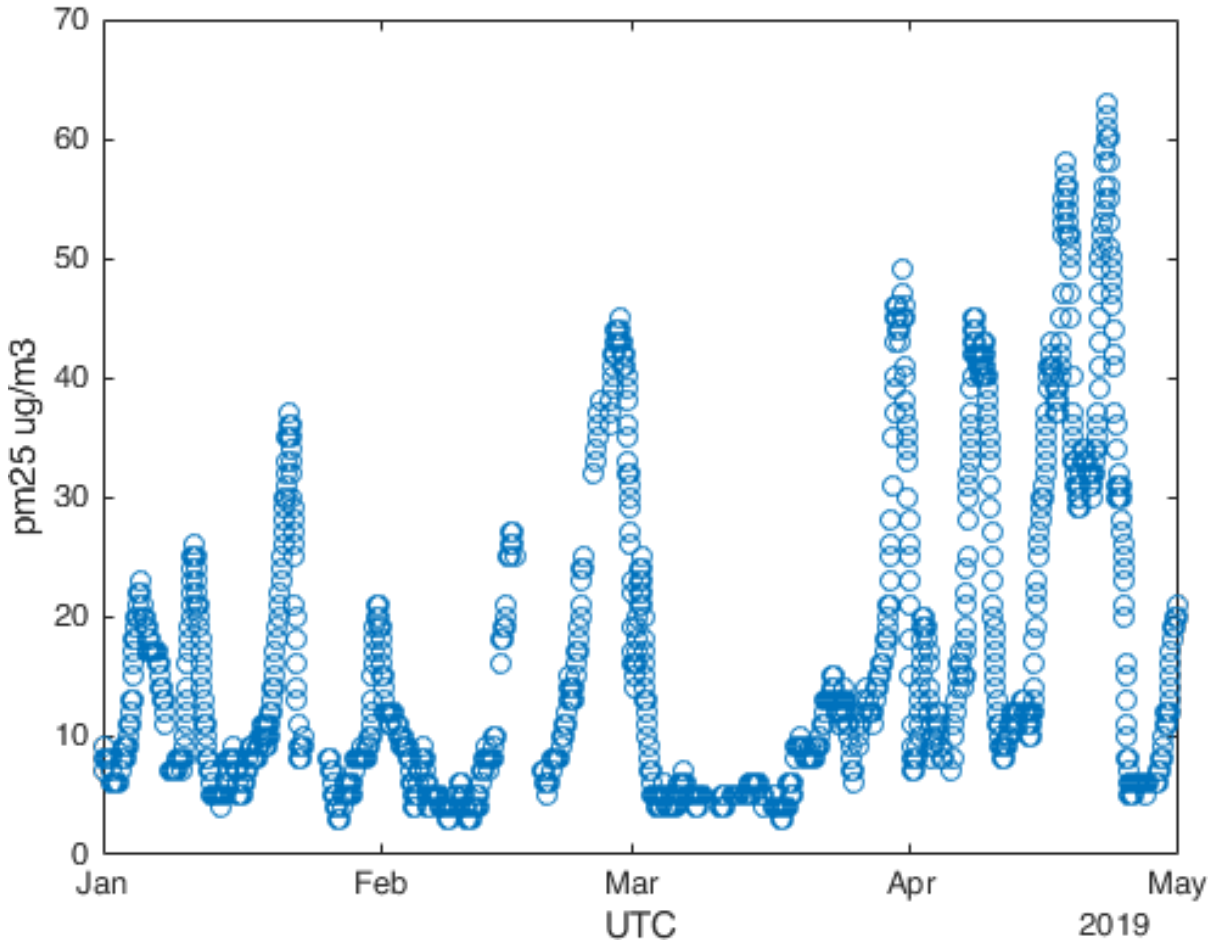
We focus on the first of the locations `London Eltham`, and we want to plot the quantity of pm25 we have measured in the whole period. For doing this we need to perform a slicing of the data. We need to find the rows that correspond to this location, we can do this by using the command

```
index = strcmp(london.location,'London Eltham');
```

at the end of this call the variable `index` will be a vector having a `1` in position `i` if `london.location(i)` is `'London Eltham'`, and a `0` otherwise. With this knowledge we can now produce a `plot` of these values by adding to the script

```
figure(1)
plot(london.utc(index),london.value(index),'o')
xlabel('UTC');
ylabel('pm25 ug/m3');
```

obtaining



Now let us repeat the same task for all the different location. We want to produce now a single plot with different subplots in which each of them has one of the Locations. Since we do not want to rewrite many times the same piece of code, we will make use of a `for` cycle

```
location = unique(london.location);

for i=1:length(location)
    index = strcmp(london.location,location{i});
    figure(2)
    subplot(5,2,i)
    plot(london.utc(index),london.value(index),'o')
    xlabel('UTC');
    ylabel('pm25 ug/m3');
    title(location{i});
end
```

The first line `location = unique(london.location);` produces, as you may guess, the unique list of locations of our table. If we ask it on the command window, we discover that these are

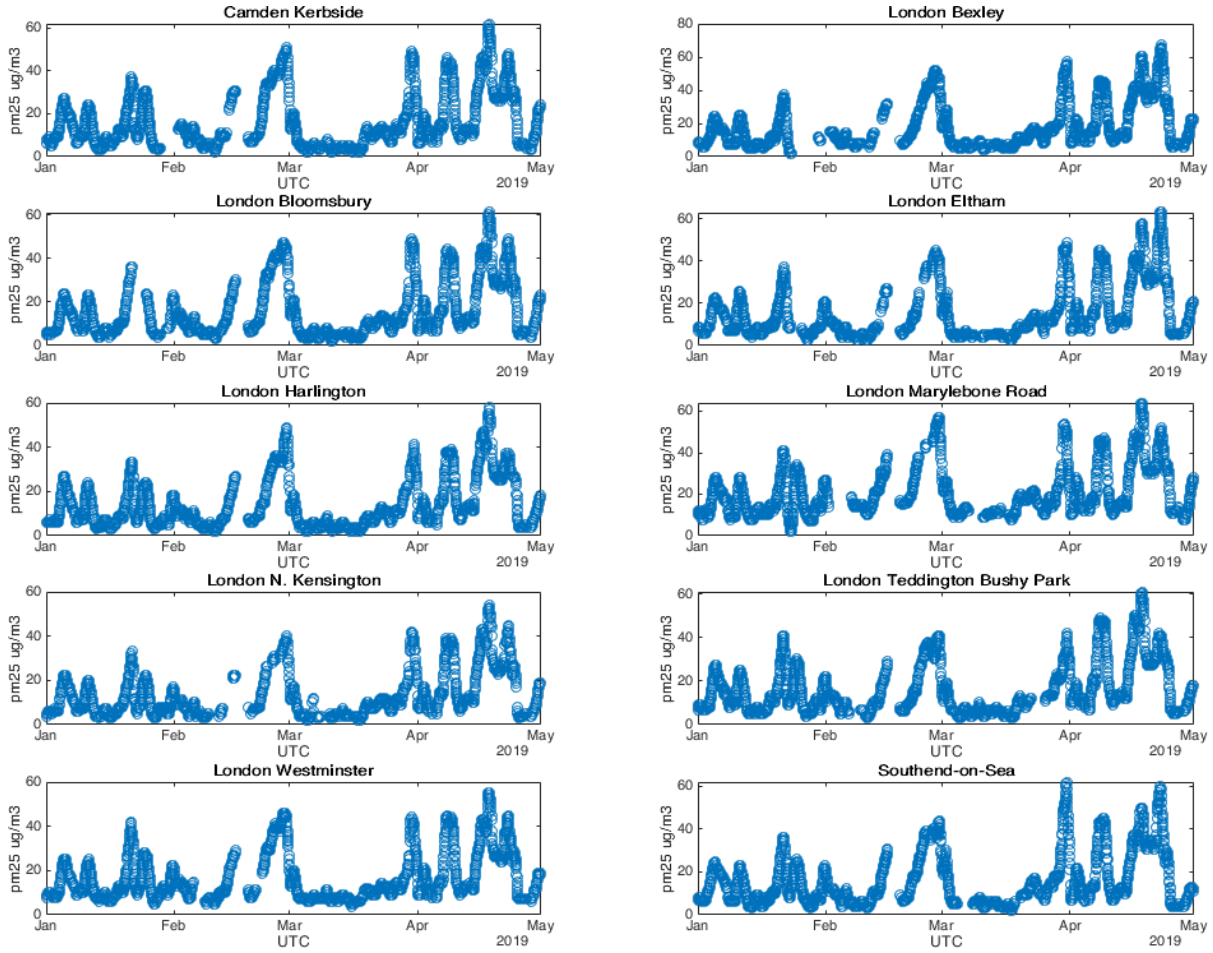
```
>> location

location =

  10×1 cell array
```

(continues on next page)

(continued from previous page)

```
    {'Camden Kerbside'              }
    {'London Bexley'                }
    {'London Bloomsbury'            }
    {'London Eltham'                }
    {'London Harlington'            }
    {'London Marylebone Road'       }
    {'London N. Kensington'         }
    {'London Teddington Bushy Park'}
    {'London Westminster'           }
    {'Southend-on-Sea'              }
```

Then we loop the code for all the unique locations and repeat the same procedure as before, with some small difference. When we look for the `index` vector we now do the comparison with each and every location by looping through the location `cell array` with the `i` index, i.e., `index = strcmp(london.location,location{i});`. Then the remaining part is pretty much the same, a part from the command `subplot` that tell us the number of panels in which we want to subdivide `figure(2)`, in this case 5 rows and 2 columns, and in which of them we are going to plot, the ith panel at cycle `i`. If we run all this code, we get:



A variant of this idea could be the one of having all the plots overlapped on the same figure to do a fast comparison

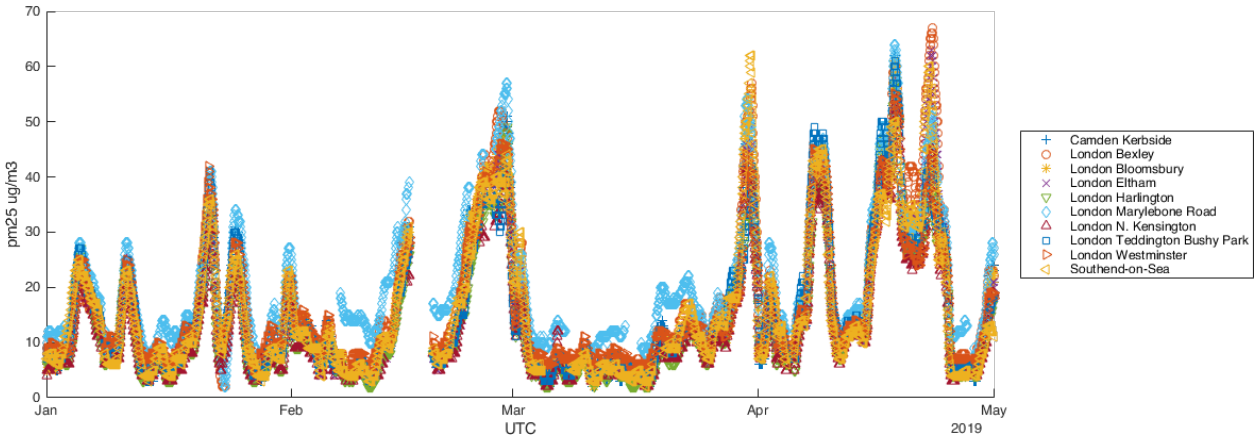
```
Markers = {'+','o','*','x','v','d','^','s','>','<'};
for i=1:length(location)
    index = strcmp(london.location,location{i});
```

(continues on next page)

(continued from previous page)

```
    figure(3)
    hold on
    plot(london.utc(index),london.value(index),Markers{i},'DisplayName',location{i})
    hold off
end
xlabel('UTC');
ylabel('pm25 ug/m3');
legend('Location','eastoutside');
```

from which we obtain



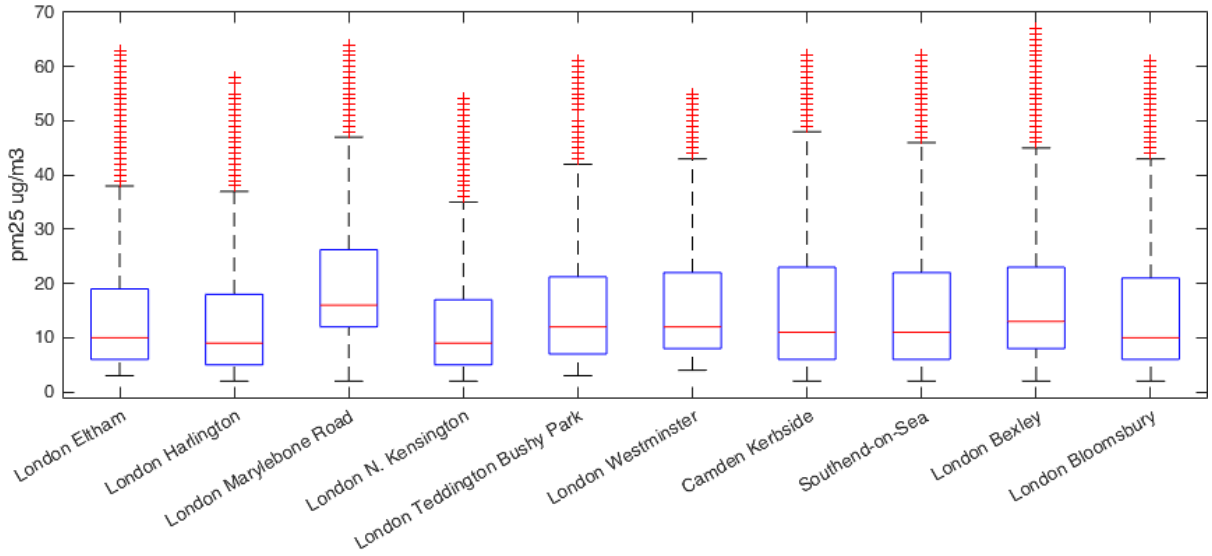
We have introduced here several new keywords,

- `hold on` retains plots in the current axes so that new plots added to the axes do not delete existing plots.
- `hold off` sets the hold state to off so that new plots added to the axes clear existing plots and reset all axes properties.
- `legend` creates a legend with descriptive labels for each plotted data series. For the labels, the legend uses the text from the `DisplayName` properties of the data series.

This visual comparison we have constructed is, however, rather inconclusive. Let us try producing a **box plot** for the different locations. This can be done with:

```
figure(4)
boxplot(london.value,london.location);
ylabel('pm25 ug/m3');
xtickangle(30)
```

that produces



The commands we have used here are

- `boxplot(x,g)` creates a box plot using one or more grouping variables contained in `g`. `boxplot` produces a separate box for each set of `x` values that share the same `g` value or values.
- `xtickangle` rotates the x-axis tick labels for the current axes to the specified angle in degrees, where 0 is horizontal. Specify a positive value for counterclockwise rotation or a negative value for clockwise rotation.

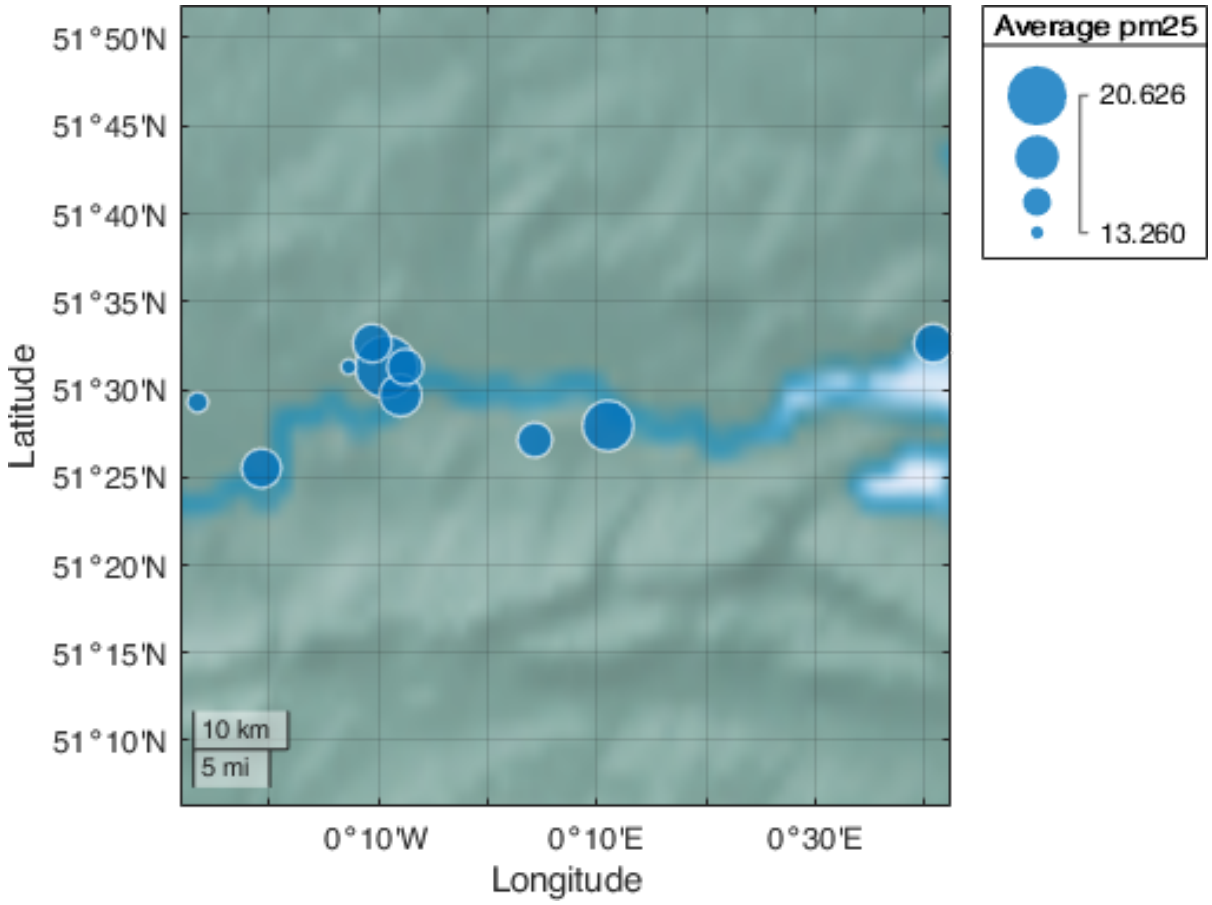
From this figure we discover that living in *Marylebone Road* is worse on average for the quantity of pm25.

Danger: The following part depends on having the Mapping Toolbox installed.

To conclude this part, let us put some of these information on a geographical map. First we need to collect the *latitudes* and *longitudes* of the different locations, then we decide that the size of the marker on the map will be given by the `mean` of the values of pm25 in that location:

```
latitude = zeros(10,1);
longitude = zeros(10,1);
average = zeros(10,1);
for i=1:10
    index = strcmp(london.location,location{i});
    latitude(i) = unique(london.latitude(index));
    longitude(i) = unique(london.longitude(index));
    average(i) = mean(london.value(index));
end
figure(5)
tab = table(latitude,longitude,average);
gb = geobubble(tab,'latitude','longitude', ...
    'SizeVariable','average');
gb.SizeLegendTitle = 'Average pm25';
geobasemap colorterrain
```

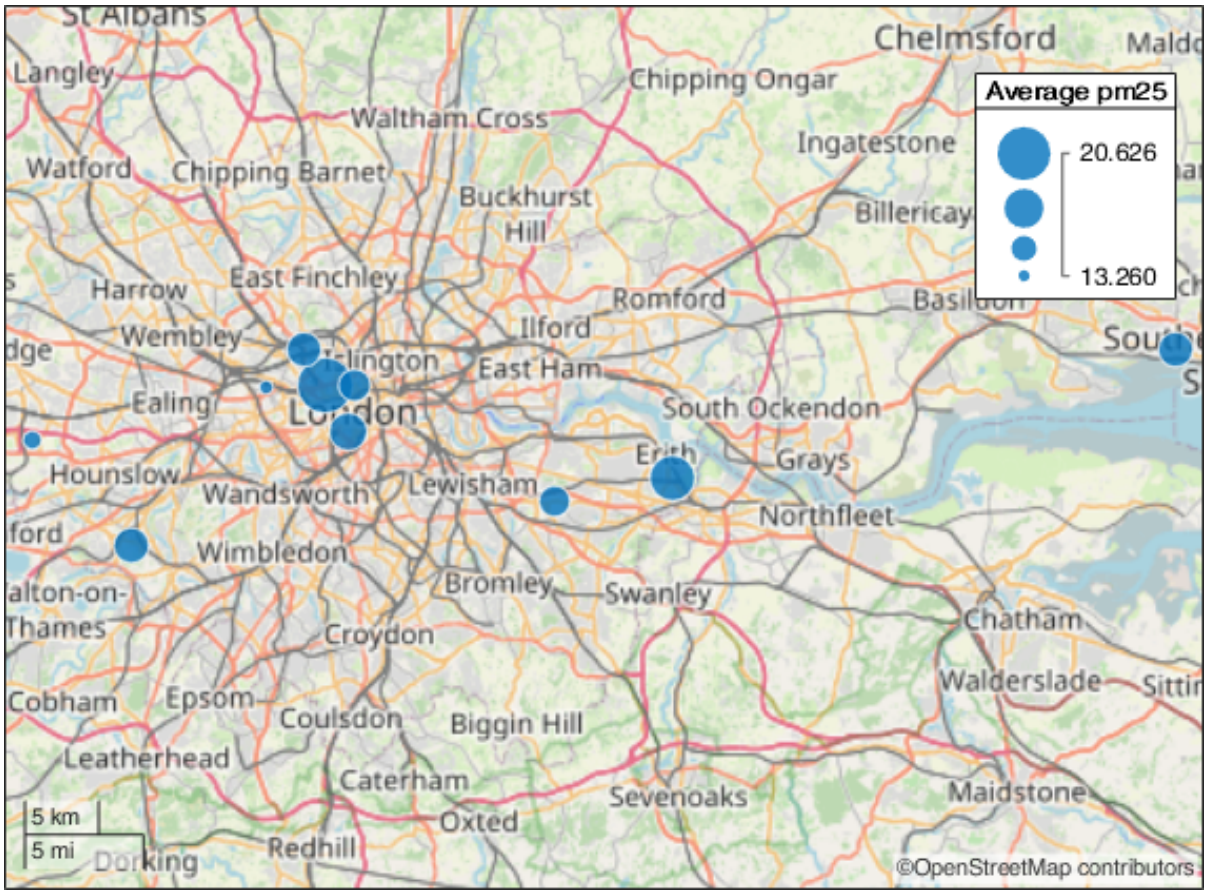
from which we get

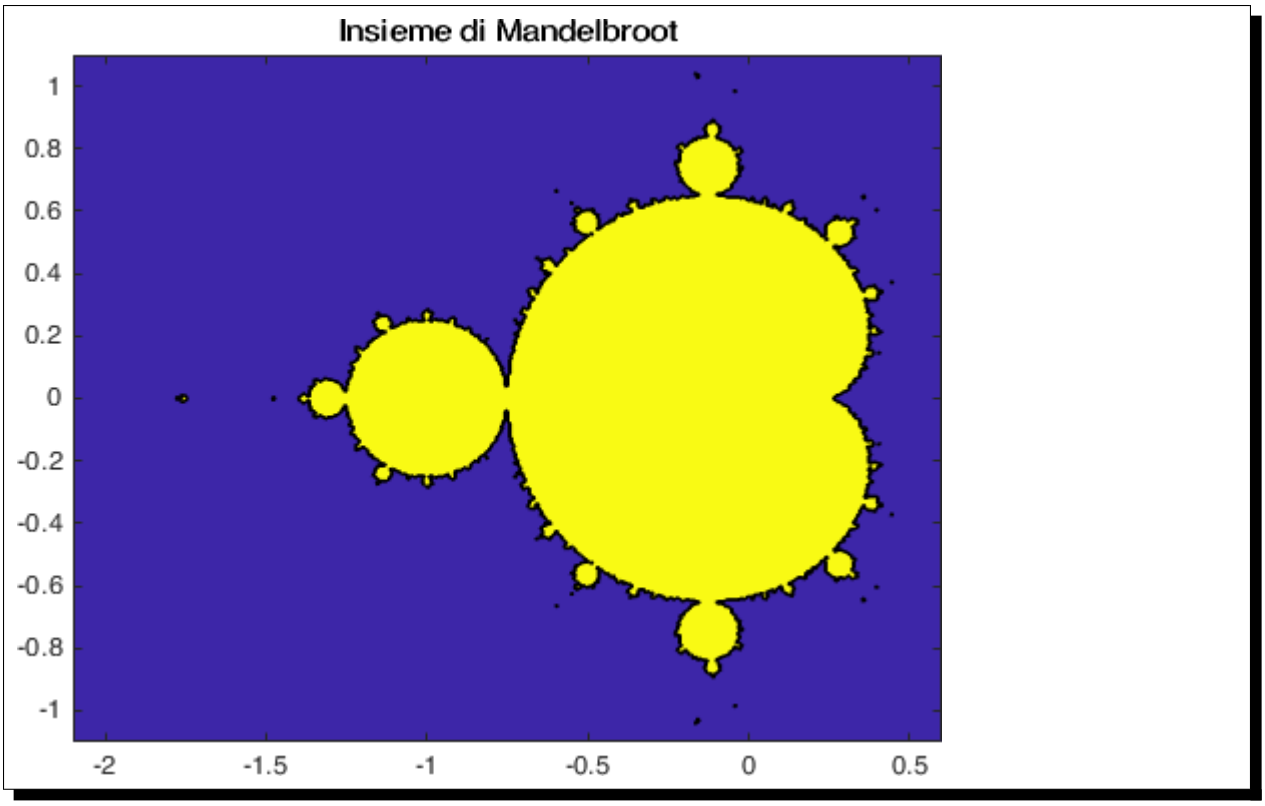


A better view using a map including the streets can be obtained by doing

```
figure(6)
gb = geobubble(tab,'latitude','longitude', ...
    'SizeVariable','average');
gb.SizeLegendTitle = 'Average pm25';
name = 'openstreetmap';
url = 'a.tile.openstreetmap.org';
copyright = char(uint8(169));
attribution = copyright + "OpenStreetMap contributors";
addCustomBasemap(name,url,'Attribution',attribution)
geobasemap openstreetmap
gb.MapLayout = 'maximized';
```

from which we obtain


St Albans
Langley
Chelmsford
Chipping Ongar
Waltham Cross
Ingatestone
Watford
Chipping Barnet
Buckhurst Hill
Billericay
Harrow
East Finchley
Romford
Wembley
Ilford
East Ham
Ealing
South Ockendon
Hounslow
Grays
Wandsworth
Lewisham
Northfleet
Wimbledon
Bromley
Swanley
Chatham
Croydon
Walderslade
Cobham
Epsom
Coulsdon
Biggin Hill
Leatherhead
Caterham
Sevenoaks
Maidstone
Oxted
Redhill
Average pm25
20.626
13.260
5 km
5 mi
©OpenStreetMap contributors



Exercise 3

Let's explore MATLAB's * plot * functions again. We try to produce a print of the whole Mandelbrot fractal. This set is the set is the set of complex numbers c for which the function $f_c(z) = z^2 + c$ does not **diverge** when iterated starting from $z = 0$, that is, the set of those points c so the sequence $f_c(0), f_c(f_c(0)), \ldots$ remains limited in absolute value. We can build a MATLAB *script* that allows us to draw an approximation of this set.

1. We use the `linspace` function to construct the set of numbers c complexes on which we want to evaluate. Since `linspace` produces one real vector for us, we need to construct two of them – one for each direction – and transform them into a set of evaluation pairs with the `meshgrid` function. A good real set to evaluate to draw the Mandelbrot set is $[-2.1, 0.6] \times [-1.1, 1.1]$.
2. Now that we have the real evaluations, we need to transform c into complex numbers. We can do this by using the `complex` function:`C = complex (X, Y)` on the pair of evaluation matrices obtained from `meshgrid`.
3. We can now implement a fixed number of iterations of the function $f_c(z) = z^2 + c$ using a `for` loop.
4. We conclude the exercise by drawing the Mandelbrot set with the function

```
contourf(x,y,double(abs(Z)<1e6))
title('Mandelbrot set')
```

which is a good chance to see what the *plot* function does `contourf` (`help contourf`).

Tip: There are many others plotting functions, but we will focus on them in the following topics, while we solve problems for which they will be useful.

2.1 Writing data to screen and to file

MATLAB provides a fairly transparent porting of C's screen printing functions (on data streams). That is, the fprintf function. For screen printing the prototype of this function is

```
fprintf(FORMAT, A, ...)
```

where FORMAT is a string that contains information about the format to be printed and `A` is an array that contains the data to be printed according to the FORMAT format. In general this is a string that can contain text accompanied by *escape* characters that tell you how to format the data contained in the `A` variable.



As described in the image, the *escape* for a formatting operator begins with the percent sign, `%`, and ends with a conversion character (Table 2.1). The conversion character is required. Optionally, you can specify an identifier, flags, field width, *precision*, and a *subtype* operator between the `%` and the conversion.

Table 2.1: Conversion characters

Carattere	Conversione
`%d` o `%i`	Intero base 10
`%f`	Floating point fixed precision
`%e`	Floating point scientific notation
`%c`	Single character
`%s`	String

An example:

```
fprintf("%f \n",pi);
fprintf("%e \n",5*10^20);
fprintf("%1.2f \n",pi);
fprintf("%1.2e \n",5*10^20);
fprintf("%c \n",'a')
fprintf("%s \n",'Ciao, mondo!')
```

In the example we have repeatedly used the `\n` characters which symbolize a newline character. Other useful characters of this type are in Table 2.2.

Table 2.2: Formatting characters

Result	String
Single quotation mark	`''`
Percent symbol	`%%`
Backslash	`\\`
Backspace	`\b`
Tab horizontal	`\t`
Tab vertical	`\v`

More information can be obtained by writing `help fprintf` in the *command line*.

CHAPTER

THREE

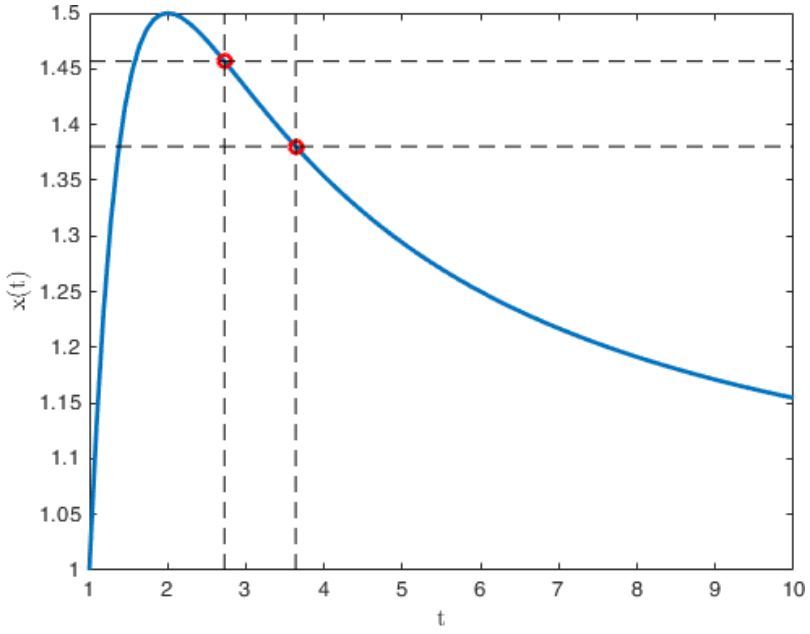
MODELING EVOLUTIONARY PROBLEMS

One of the ways in which mathematics is used to translate experimental measurements into an understanding of the underlying regulating mechanisms is represented by **ordinary differential equations**. Indeed if we consider the values of variables these do not provide themselves insight on the underlying phenomena we want to describe. The fundamental insight is that is *the change in the magnitude* of a variable *as a function of time* that is the important quantity. These changes can be often be described in terms of differential equations.

The first class of problems we will address in this course is then their numerical solution with MATLAB. To arrive at this we will start by (quickly) recalling the mathematical background.

3.1 Ordinary differential equations

Let us suppose that we have a variable x that depends on time t, we can represent its evolution on a graph



There are intervals during which x increases and others during which it decreases. We can denote the **slope** of the change as

$$\frac{\Delta x}{\Delta t} = \frac{x(t_2) - x(t_1)}{t_2 - t_1} = \frac{x(t_1 + \Delta t) - x(t_1)}{\Delta t}, \quad \Delta t = t_2 - t_1,$$

when we define it in this way, the *slope* is a function of the selected interval. We are the interested in what happens as we

let the time interval decrease by approaching zero, i.e., $\Delta t \to 0$,

$$\lim_{\Delta t \to 0} \frac{\Delta x}{\Delta t} = \lim_{\Delta t \to 0} \frac{x(t_1 + \Delta t) - x(t_1)}{\Delta t} = x'(t) = \frac{\mathrm{d}x}{\mathrm{d}t} = \dot{x}(t),$$

that is, we are interested in its **derivative**. Specifically, we are interested in making **predictions** about phenomena that are subject to change, e.g.,

- the rate at which the food supply for a given specie is related to the growth of its population,
- the variation of a drug concentration in an organ,
- the rate of decay of a radioactive substance.

Thus we are particularly interested in equations that tell us how this rate at which a given quantity changes when it is related to some function of the quantity itself. In mathematical terms these are equations of the form

$$x'(t) = f(t, x(t)), \quad t \in [t_0, t_f].$$

These type of equations are called **ordinary differential equations**. Their solution cannot be determined uniquely without employing some **outside condition**. This is typically an *initial value*, e.g., the quantity of radioactive material at the beginning of the decay or the number of members of our population.

There is a very rich mathematical theory behind these objects that is involved in

1. having strategies useful for the development of models of physical phenomena;
2. discovering whether the model we (or our *friendly neighborhood mathematician*) have just devised is **well-posed** (are there solutions? are these unique? do the solutions correspond to the phenomenon we were modeling?);

But don't worry, what we are interested in here is using MATLAB to investigate them, and approximate their solution . So we will assume that you have got your ODE from a good dealer, and avoid investigating the theory.

We will start from **an example** doing the whole analysis from top to bottom, then you will have the opportunity to work on a **gallery of problems** testing what we have learned.

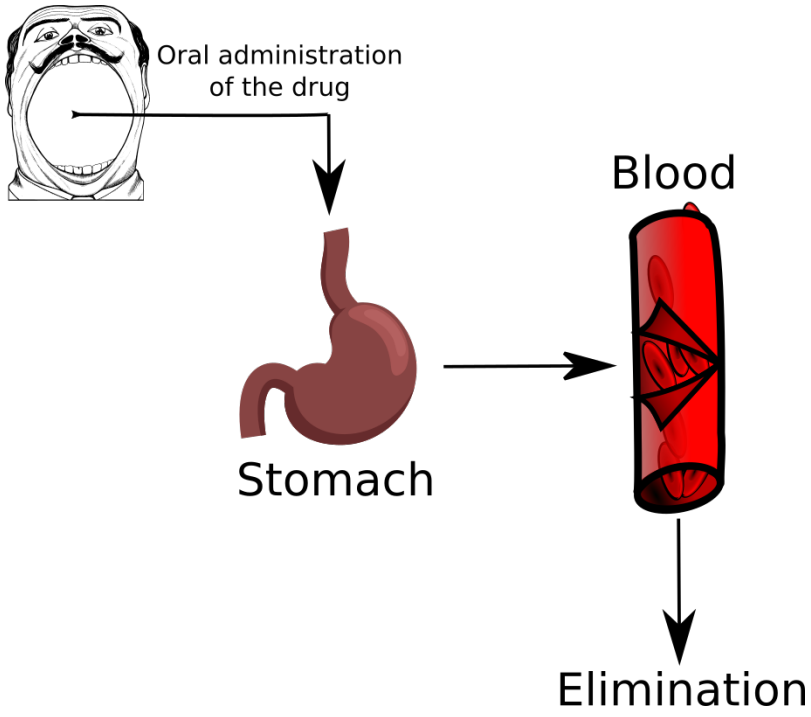
3.2 A completely worked out example

The model we want to study is an of example of **compartment analysis** related to the time evolution of a drug concentration in an organ. We let now $x(t)$ denote the amount of substance in some compartment at time t. We can then compute the change in the quantity $x(t)$ in terms of the amount of the quantity flowing into and out the compartment, i.e., we are assuming that the substance does not "disappear" in the process,

$$\frac{\mathrm{d}x(t)}{\mathrm{d}t} = \text{input rate } - \text{output rate},$$

this principle is based on the **law of conservation of mass** and is known as the *Balance Law*.

The first example we look at is a simple process of drug administration through the stomach and the blood



We enter in the system y_0 unit of drugs (*initial condition*), then the quantity of drug that is in the stomach (*first compartment*) is denoted by $y_1(t)$. From here it can only go to the blood (*second compartment*) with a transition rate of k_1. We denote with $y_2(t)$ the quantity of drug in the second compartment, from here then the drug is eliminated through some metabolic process at rate k_e. Let us write down the differential model for this case

$$\begin{cases} \dot{y}_1(t) = -k_1 y_1(t), & y_1(0) = y_0, \\ \dot{y}_2(t) = +k_1 y_1(t) - k_e y_2(t), & y_2(0) = 0, \end{cases} \quad t \in [0, t_f],$$

we have written down a system of two differential equations for the quantity of drug in the two compartments.

We now use MATLAB to get a **numerical solution** of this system. In this case a numerical solution is nothing more than an evaluation of the two functions $y_1(t)$, and $y_2(t)$ over a grid of time values

$$0 = t_1 < t_2 < ... < t_n = t_f.$$

Let us start by creating a `matlab script` called `drugdelivery.m`.

```
%% Drug Delivery
% This script will be used to solve a simple drug-delivery model in the form
% of two linear coupled odes

clear; clc; close all; % Clean the memory
```

In MATLAB you can specify any number of coupled ODE equations to solve, and at least in principle the number of equations is only limited by the available computer memory. In our case we can specify our *dynamics* by using a `function handle`, by adding to the previous script

```
k1 = 0.9776;       % Transition rate between stomach and blood
ke = 0.2213;       % Eliminatio rate from the blood
A = [ -k1 0; k1 -ke]; % Two by two matrix
f = @(t,y) A*y;    % Right-hand side of the ODE
```

We have firs fixed the values of the two constants for the model. Then we have rewritten the right-hand side of the model

as a matrix vector product

$$\begin{bmatrix} \dot{y_1}(t) \\ y_2(t) \end{bmatrix} = \begin{bmatrix} -k_1 & 0 \\ +k_1 & -k_e \end{bmatrix} \begin{bmatrix} y_1(t) \\ y_2(t) \end{bmatrix} \Leftrightarrow \dot{\mathbf{y}} = A\mathbf{y} = f(t, \mathbf{y}).$$

in a `function handle` form. That is, we can call `f(t,y)` to evaluate the dynamics.

Warning: The dynamics of the system must always be expressed with the arguments in this order: first the independent variable (t), then the function ($\mathbf{y}$). MATLAB expects it to be this way!

The other two data from the system that we need are the *initial condition*, and the maximum time t_f. We can add them to the script by doing

```
y0 = [600;0]; % firt component is the initial condition for y1, the second for y2
t0 = 0;
tf = 6;
```

Now we have specified all the needed data and we can use one of the MATLAB **ODE integrator** to solve the system

```
[T,Y] = ode45(@(t,y) f(t,y),[t0,tf],y0);
```

After this call had been executed the variable `T` will contain the values

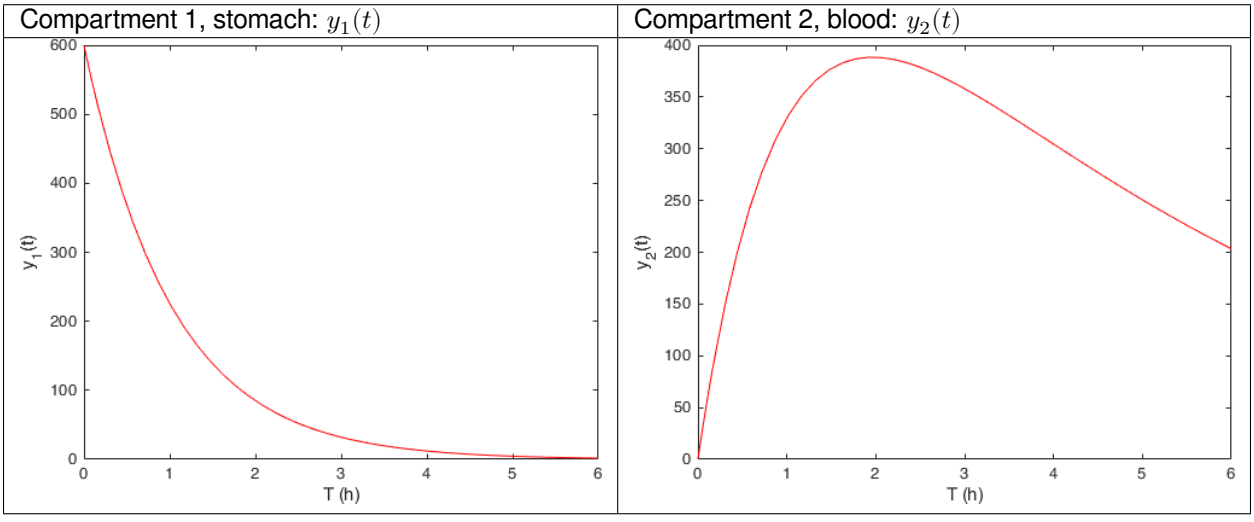
$$0 = t_1 < t_2 < ... < t_n = t_f$$

on which we have approximated the solution, while Y will be a lenght$(T) \times 2$ matrix containing the approximation of the two solutions on each time step. We can visualize what we have obtained by doing

```
figure(1)
plot(T,Y(:,1),'r-')
xlabel('T (h)')
ylabel('y_1(t)')
figure(2)
plot(T,Y(:,2),'r-')
xlabel('T (h)')
ylabel('y_2(t)')
```

and obtaining the two figures

Table 3.1: Simple drug delivery solutions



From which we observe somehow the expected behavior. The quantity of drug in the stomach (first compartment on the left) decreases with the time, while the quantity in the bloodstream starts (second compartment on the right) to increase, but since it is also eliminated from there with rate k_e it shows that hump.

Tip: `ode45` performs well with *most* ODE problems and should generally be your first choice of solver. However, `ode23`, `ode78`, `ode89` and `ode113` can be more efficient than `ode45` for problems with looser or tighter accuracy requirements.

Some ODE problems exhibit **stiffness**. Stiffness is a term that defies a precise definition, but in general, stiffness occurs when there is a difference in scaling somewhere in the problem. For example, if an ODE has two solution components that vary on drastically different time scales, then the equation might be stiff. You can **identify a problem as stiff** if *nonstiff* solvers (such as `ode45`) are unable to solve the problem or are extremely slow. If you observe that a nonstiff solver is very slow, try using a stiff solver such as `ode15s` instead.

Let us **summarize** the different steps we need for a program solving an ODE (a system of ODEs),

1. we *implement* in either a *function* or a *function handle* the **dynamics of the system**;
2. we create a vector containing the **initial condition**;
3. we fix the time interval `[t0,tf]` on which we want to solve our problem;
4. we call an ODE solver (typically `ode45` or `ode15s`) using all these data.

Remark

Instead of just `[t0,tf]` we could pass to the integrator the whole vector of time steps we want to use. A canonical way for doing so is, e.g., by generating a linearly spaced vector such has

```
T = linspace(t0,tf,100);
```

This can be useful if we want the (approximate) solution to be evaluated at certain time steps.

If we are uncertain at the beginning on what time steps we may be interested MATLAB offers us a routine to query the solution at any point between `[t0,tf]`. To use it we need to slightly change the way in which we call the ODE solver:

```
sol = ode45(@(t,y) f(t,y),[t0,tf],y0);
```

Now `sol` will be a `struct` variable, in the case of this example:

```
sol =

  struct with fields:

    solver: 'ode45'
   extdata: [1×1 struct]
         x: [1×20 double]
         y: [2×20 double]
     stats: [1×1 struct]
     idata: [1×1 struct]
```

The information that was contained before in the `T` vector is in `sol.x`, while what we had stored in the `Y` matrix is in `sol.y`.

Now let us say that we want to know the quantity of the drug in the bloodstream after 2 hours, we find it by doing

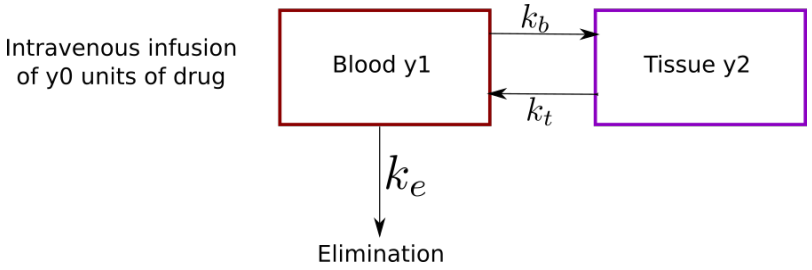
```
drug_in_blood = deval(sol,2,2);
```

The function `deval` we have used takes as a first argument the `struct` coming from the ODE solver, as second the point $t \in [t_0, t_f]$ on which we want the solution, and as last argument what component of the solution we want. In this case we have queried for the vale in the second *compartment*.

To see if we have understood how to work with this type of models you can try and solve the following variations on the theme, that have either different interactions, or a larger number of *compartments*.

Exercise 4 (Drug administration through blood and tissue)

In this model, there are again only two *compartments*. The first one is the bloodstream into which the drug is injected and the second one is the tissue where the drug has the desired effect. The blood takes a part of drug at the rate of k_b onto tissue while a fraction of it gets eliminated from the stream with elimination rate of k_e.



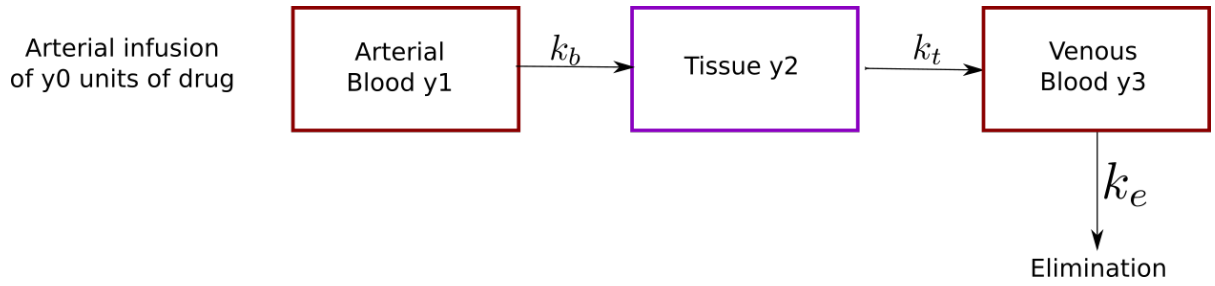
1. Write down the set of two differential equations for this model,
2. Use MATLAB to simulate the solution for $y_0 = 500$, $k_b = 0.5$, $k_e = 0.05$ and $k_t = 0.25$ again on a 6 h interval.

ODEs

$$\dot{\mathbf{y}} = \begin{bmatrix} -(k_b + k_e) & k_t \\ k_b & -k_t \end{bmatrix} \mathbf{y}, \quad \mathbf{y} = \begin{bmatrix} y_1(t) \\ y_2(t) \end{bmatrix}, \quad \mathbf{y}_0 = \begin{bmatrix} y_0 \\ 0 \end{bmatrix}.$$

Exercise 5 (…through arterial blood, tissue and venous blood)

Another model can be obtained by considering that the blood flow in cardiovascular system is one directional. Thus we can administer our drug through arterial blood:



The consumption of drug by arterial blood towards tissue has a rate k_b, from tissue compartment to the venous bloodstream has rate k_t. Then the kidneys and liver excrete the drug from the bloodstream with rate k_e.

1. Write down the set of three differential equations for this model,
2. Use MATLAB to simulate the solution for $y_0 = 500$, $k_b = 0.9776$, $k_e = 0.2213$, and $k_t = 0.3293$ again on a 6 h interval.

ODEs

$$\dot{\mathbf{y}} = \begin{bmatrix} -k_b & 0 & 0 \\ k_b & -k_t & 0 \\ 0 & k_t & -k_e \end{bmatrix} \mathbf{y}, \quad \mathbf{y} = \begin{bmatrix} y_1(t) \\ y_2(t) \\ y_3(t) \end{bmatrix}, \quad \mathbf{y}_0 = \begin{bmatrix} y_0 \\ 0 \\ 0 \end{bmatrix}.$$

3.3 Chemical kinetics

Another class of problems that ends up producing ODEs are the one coming from **chemical kinetics**, as you sure know better than me, this is the study of the rates of chemical reactions. The following examples focus on a couple of reaction mechanisms for which we can find ODEs that will allow us to calculate the concentration of the different species that take part of the reaction during the whole reaction time.

Oscillating reactions

Let us consider the following chemical reactions:

$$A + X \xrightarrow{k_1} 2X,$$
$$X + Y \xrightarrow{k_2} 2Y,$$
$$Y \xrightarrow{k_3} B,$$

in which each reaction step refers to the molecular mechanism by which the reactant molecules combine produce some intermediate products before yielding the final one. In the first step a molecule of species A combines with a molecule of species X yielding two molecules of species X. This step consumes molecules of species A in favor of molecules of species X at a rate that is proportional to the product of the concentrations of A and X.

If we write down the system of differential equations for this formulation we get

$$\begin{cases} \frac{\mathrm{d}[A]}{\mathrm{d}t} = -k_1[A][X], \\ \frac{\mathrm{d}[X]}{\mathrm{d}t} = +k_1[A][X] - k_2[X][Y], \\ \frac{\mathrm{d}[Y]}{\mathrm{d}t} = +k_2[X][Y] - k_3[Y], \\ \frac{\mathrm{d}[B]}{\mathrm{d}t} = -k_3[Y], \end{cases}$$

that are four differential equations in four variables.

It is **important to observe** that for this case, the equations are **nonlinear**: they depend on the product of variables, i.e., we cannot represent them as we have done for the other examples in matrix form.

To solve this system we will make some assumptions, and observation

1. the concentration of the reactant A is held constant: somebody is feeding the system at a rate equal to k_1 a supply of A,
2. $\frac{\mathrm{d}[A]+[X]+[Y]+[B]}{\mathrm{d}t} = 0$, so $[A] + [X] + [Y] + [B]$ is constant,
3. at the beginning there is no $[B]$, thus $[A] + [B] + [X] + [Y] = [A]_0 + [X]_0 + [Y]_0$,

This means that we do not need to solve the last equation, we can just compute $[B] = [X]_0 - X + [Y]_0 - Y$, and the first equation is also solved because we have decided to make the quantity of $[A]$ constant. We have thus reduced to solving the system

$$\begin{cases} \frac{\mathrm{d}[X]}{\mathrm{d}t} = +k_1[A]_0[X] - k_2[X][Y], \\ \frac{\mathrm{d}[Y]}{\mathrm{d}t} = +k_2[X][Y] - k_3[Y], \end{cases}$$

We can write the dynamics in MATLAB again as a *function handle*, after we have defined the constants `k1`, `k2`, and `A0` by writing

```
f = @(t,y) [k1*A0*y(1)-k2*y(1)*y(2); k2*y(1)*y(2)-k3*y(2)];
```

3.4 A simple epidemiological model

We consider again a compartmental model, but this time for the spread of an epidemics in a population. We first partition the host population into three compartments:

- S susceptible hosts,
- I infectious hosts,
- R recovered hosts. The objective of our modeling effort is again to track the number of hosts in each of the three compartments at any given time t, that is we want to know the functions $S(t)$, $I(t)$, and $R(t)$.

By applying again the conservation principle, we know that the net change of the number of hosts in a compartment can be expressed as the difference between the number coming into the compartment and the number leaving it during the time interval under consideration:

$$\begin{aligned}\Delta S(t) =&\text{"new susceptible"} + \text{"transfer from "}R - \text{"new infections"} - \text{"removal from "}S,\\ \Delta I(t) =&\text{"new infections"} - \text{"transfer into "}R - \text{"removal from "}I,\\ \Delta R(t) =&\text{"transfer from "}I - \text{"transfer into "}S - \text{"removal from "}R,\end{aligned}$$

then we divide both sides by Δt, and we let it go to 0, $\Delta t \to 0$, we find the derivatives of the tree functions $S(t)$, $I(t)$, and $R(t)$ on the left-hand side, and the *transfer rate* on the right-hand side. Succinctly:

$$\begin{cases} S'(t) = -\lambda IS, & S(0) = S_0 > 0\\ I'(t) = \lambda IS - \gamma I, & I(0) = I_0 > 0,\\ R'(t) = \gamma I, & R(0) = R_0. \end{cases}$$

Where have **assumed** that

- transmission occurs through direct contact between hosts;
- the *incidence rate*, that is the number of new infections per unit time, can be expressed as $\lambda I(t)S(t)$ for a given *transmission coefficient* λ,
- the *recovery rate* can be written as $\gamma I(t)$ for some constant rate γ,
- the population is fixed, there is no possibility of being reinfected after healing.

These assumptions will be much or less reasonable depending on the the infectious disease. For our illustrative needs, this simple model will be good enough.

Now we have our system of ODEs together with the initial condition, so given some values for the constants λ, γ, S_0 and I_0 we could jump right in and apply one of the numerical integrator from MATLAB to get the solution. **However** we can slightly simplify the model by observing that

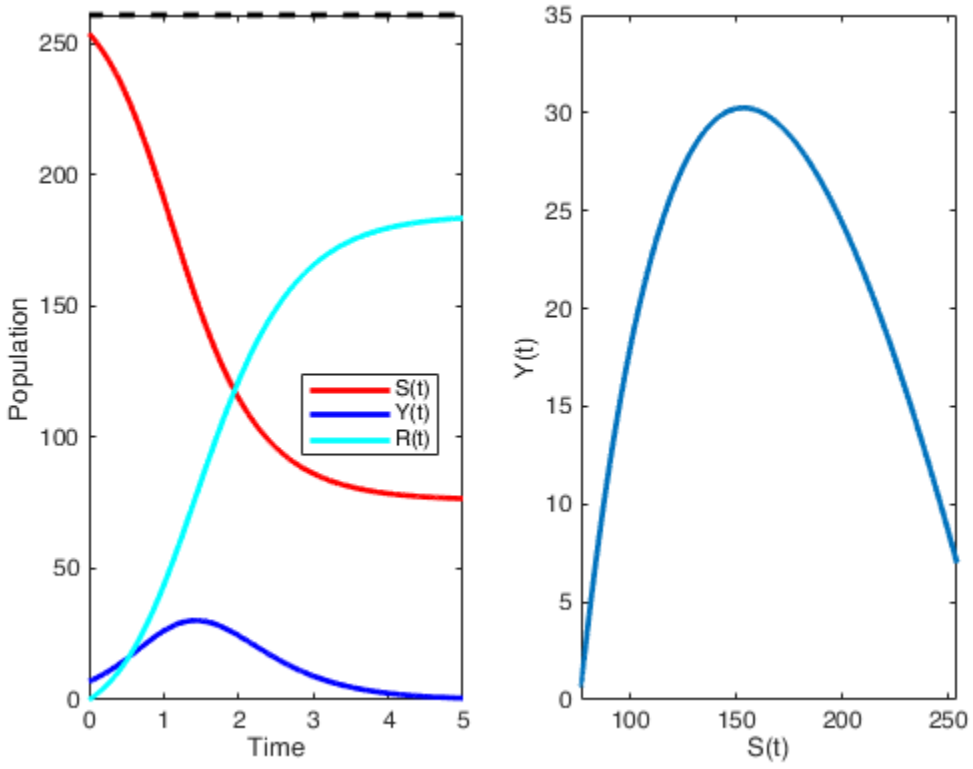
$$N = S(t) + I(t) + R(t) = S(0) + I(0) + R(0) = S(0) + I(0),$$

therefore $dN/dt = 0$, and thus we don't need to solve the equation for $R(t)$, since

$$R(t) = N - S(t) - I(t).$$

Now let us **build a simulation**.

```
clear; clc; close all

N = 261;          % Size of the population
I0 = 7;           % Initially infected
S0 = N-I0;        % Number of susceptible individuals
lambda = 0.0178; % Transmission coefficient
gamma = 2.73;     % Recovery rate

% We order the variables as y(t) = [S(t),I(t)]
f = @(t,y) [-lambda*y(1)*y(2); lambda*y(1)*y(2)-gamma*y(2)];
[T,Y] = ode45(@(t,y) f(t,y),linspace(0,5,1000),[S0;I0]);

% We plot the three curves on the same graph
figure(1)
subplot(1,2,1)
plot(T,Y(:,1),'r-',...
    T,Y(:,2),'b-',...
    T,N-Y(:,1)-Y(:,2),'c-',...
    T,N*ones(size(T)),'k--','LineWidth',2);
xlabel('Time')
ylabel('Population')
legend({'S(t)','Y(t)','R(t)'},'Location','best')
axis tight
subplot(1,2,2)
plot(Y(:,1),Y(:,2),'LineWidth',2);
xlabel('S(t)');
ylabel('Y(t)');
```



The parameters in this example have been obtained from the *great plague in Eyam*, a village near Sheffield in England

from 1665-1666 (possibly a secondary outbreak from the *Great Plague of London*). The Plague was survived only by 83 people of an original population of 350. Observe that the data here starts from a smaller population, i.e., we are neglecting the initial insurgence, $S(0) = 254$, $I(0) = 7$.

The infective period of the bubonic plague can be estimated to be around 11 days, and this gives a value of $\lambda = 0.0178$ and a value of $\beta = 2.73$. To obtain these values we have used a couple of mathematical observations. If we sum the two equations for $S'(t)$ and $I'(t)$ we find

$$(S + I)' = -\alpha I < 0,$$

thus this a decreasing positive function, henceforth it has a finite limit, and since $\lim_{t\to+\infty} I(t) = 0$, this limit is equal to the total number of susceptible hosts at the end of the epidemics. By performing some integrals (out of our scope) one finds that

$$\log \frac{S_0}{S_\infty} = \frac{\lambda}{\gamma}[1 - \frac{S_\infty}{N}] = \mathcal{R}_0[1 - \frac{S_\infty}{N}],$$

where $\mathcal{R}_0$ is called the basic reproduction number. As you may have heard lately, then to prevent the occurrence of an epidemic it is then necessary to reduce $\mathcal{R}_0$ below the threshold of 1.

As an **exercise** you can try to play around with the parameters of this model to generate the different outcomes of an epidemic obeying it.

CHAPTER

FOUR

FITTING DATA TO MODELS

Parametric fitting involves finding coefficients (parameters) for one or more models that you want to fit to some experimental data.

In general we assume data to be statistical in nature, this means that we can assume it to be divided into two components:

$$\text{``data''} = \text{``true value''} + \text{``statistical error''},$$

the other fundamental part is that we are assuming the deterministic component to be given by a model for which the random component is described as an error associated with the data:

$$\text{``data''} = \text{``model''} + \text{``error''}.$$

The model is a function of the independent data (predictor) and one or more coefficients, that are the quantities we want to compute. The error represents random variations in the data. For doing a mathematical analysis one usually assumes that they follow a specific probability distribution - in most of the case Gaussian. The source of the error can be varied, but it is always present, errors are bound to happen when you are dealing with measured data.

4.1 Least square approach

To write down formally the idea we have discussed we can express a general **nonlinear regression model** as

$$Y_n = f(x_n, \theta) + Z_n,$$

where f is the model we want to *fit*, x_n is a vector of the associated variable, Y_n are the attained measurements, θ is a vector of parameters defining the model, and Z_n is the error term.

To be more practical and less formal, let us start with the tale of a certain Count Rumford of Bavaria (see [BW88]). He was one of the early experimenters on the physics of heat. In 1798 he performed the following experiment, he heated a cannon barrel to a temperature of 130° F and then let it cool to the room temperature of 60° F while taking measure of temperature at different time intervals:

```
data = [ 4 126 % Time (min) and Temperature (°F)
  5 125
  7 123
  12 120
  14 119
  16 118
  20 116
  24 115
  28 114
  31 113
```

(continues on next page)

(continued from previous page)

```
  34 112
  37.5 111
  41 110
]
```

To interpret these data we can use Newton's law of cooling, which states that

$$\frac{\partial f}{\partial t} = -\theta(f - T_0),$$

where T_0 is the ambient temperature. This is one of the few differential equations we actually know how to solve, and indeed we can express

$$f(t, \theta) = T_0 + (T_f - T_0)e^{-\theta t} = 60 + 70e^{-\theta t}.$$

This model now depends *nonlinearly* on just one parameter θ that is the one that we have to fit to our data. Mathematically this means that we want to find θ such that

$$\min \sum_{j=1}^{\#\text{ of data}} |f(t_j, \theta) - T_j|^2,$$

that is we want to solve a **least square** problem. Typically, together with the previous *objective function* we have also some *constraints* on the parameters. For example in this case we may want to impose the constraint of having $\theta > 0$.

To solve these type of problems MATLAB offers a function called `fit`. Let us look at its usage:

```
fo = fitoptions('Method','NonlinearLeastSquares',...
                'Lower',[0],...
                'Upper',[Inf],...
                'StartPoint',[1]);
ft = fittype('60 + 70*exp(-a*x)','options',fo);
[curve,gof] = fit(data(:,1),data(:,2),ft);
```

1. First we have set up some options for the `fit` function:

- in the `'Method'` field we have selected the `'NonlinearLeastSquares'` options. This tells MATLAB that we want to solve the problem in the mathematical formulation given above.
- The `'Lower'` and `'Upper'` keywords denote the bound on the parameters in the order in which they appear in the model. In our case we have just one parameter θ so we request that `0 \leq \theta \leq \infty`, that is $\theta \geq 0$.
- Finally, with the keyword `'StartPoint'` we tell the algorithm what is the starting point of the *iterative procedure* producing the solution. **Having good starting points for the parameters could be crucial!**.

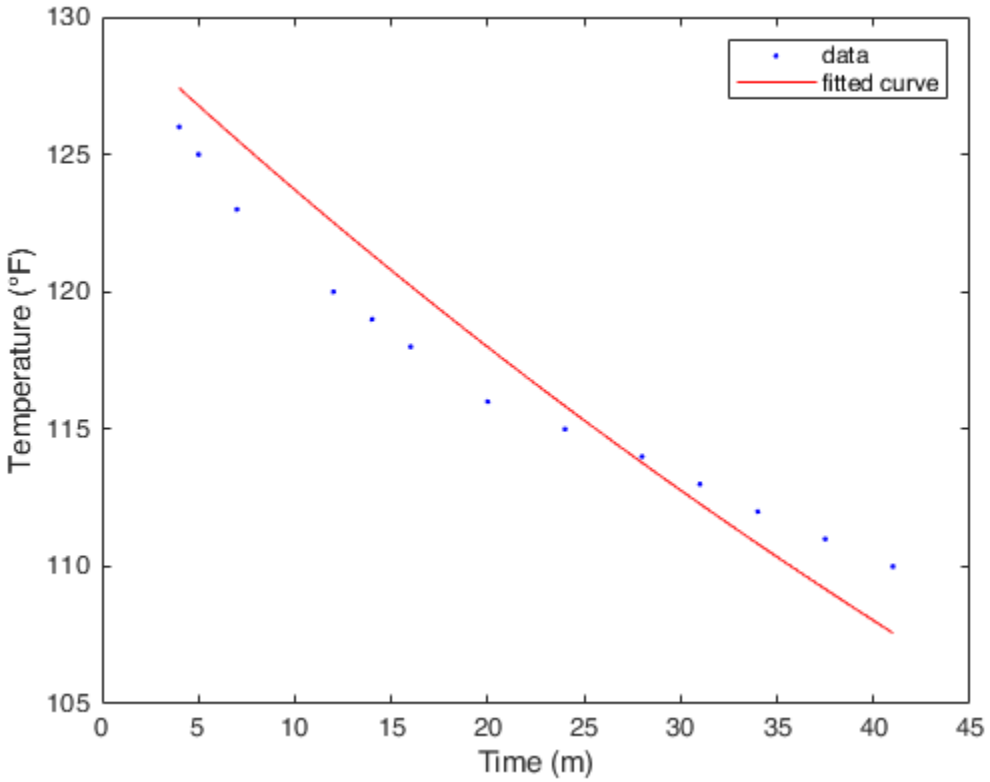
1. Then we fix the model we want to fit, the variable is always denote as `x` while the other letters appearing in the expression are interpreted as the parameters
2. We finally launch the *fit* operation with the command of the same name by passing to it the data and the model we have defined.

```
data = [ 4 126 % Time (min) and Temperature (°F)
  5 125
  7 123
  12 120
  14 119
  16 118
  20 116
```

(continues on next page)

(continued from previous page)

```
  24 115
  28 114
  31 113
  34 112
  37.5 111
  41 110
];
fo = fitoptions('Method','NonlinearLeastSquares',...
                'Lower',[0],...
                'Upper',[Inf],...
                'StartPoint',[1]);
ft = fittype('60 + 70*exp(-a*x)','options',fo);
[curve,gof] = fit(data(:,1),data(:,2),ft);
% We plot the results
figure(1)
plot(curve,data(:,1),data(:,2))
xlabel('Time (m)');
ylabel('Temperature (°F)');
```



The other information we can extract from the fit are contained in the two outputs of the `fit` function, namely:

```
disp(curve)
disp(gof)
```

```
     General model:
     curve(x) = 60 + 70*exp(-a*x)
```

```
     Coefficients (with 95% confidence bounds):
       a =    0.009416  (0.0085, 0.01033)
```

```
           sse: 44.1558
       rsquare: 0.8682
           dfe: 12
    adjrsquare: 0.8682
          rmse: 1.9182
```

Exercise 6 (Puromycin)

We use some data on the "velocity" of an enzymatic reaction. Specifically we have measured the number of counts per minute of radioactive product from the reaction as a function of substrate concentration in parts per million (ppm). From these counts the initial rate, or "velocity", of the reaction was calculated (counts/min^2). The experiment was conducted once with the enzyme treated with Puromycin and once with the untreated enzyme.

The *velocity* is assumed to depend on the substrate concentration according to the Michaelis-Menten, i.e.,

$$f(x, \theta) = \frac{\theta_1 x}{\theta_2 + x},$$

we want to

1. Compute the parameters θ_1 and θ_2 in the two cases (with and without using Puromycin),
2. Verify the hypothesis on the fact that the ultimate velocity parameter θ_1 should be affected by the introduction of the Puromycin, but not the half-velocity parameter θ_2.

```
data = [ % substrate treated untreated
  0.02 76 67
  0.02 47 51
  0.06 97 84
  0.06 107 86
  0.11 123 98
  0.11 139 115
  0.22 159 131
  0.22 152 124
  0.56 191 144
  0.56 201 158
  1.10 207 160
  1.10 200 NaN
 ]
```

Exercise 7 (Growth of leaves)

Try to find parameters for the Richards model for the growth of leaves, i.e.,

$$f(x, \theta) = \frac{\theta_1}{(1 + \theta_2 e^{-\theta_3 x})^{1/\theta 4}},$$

on the following data

```
data = [ % Time (days) Leaf length (cm)
  0.5 1.3
  1.5 1.3
  2.5 1.9
```

(continues on next page)

(continued from previous page)

```
  3.5 3.4
  4.5 5.3
  5.5 7.1
  6.5 10.6
  7.5 16.0
  8.5 16.4
  9.5 18.3
  10.5 20.9
  11.5 20.5
  12.5 21.3
  13.5 21.2
  14.5 20.9
]
```

4.2 Fitting data to a differential model

Until now we have always assumed to explicitly know the model. Nevertheless, such models often come as the solution of a differential equation. As we have hinted in the last topic, an explicit solution of a differential equation is usually hard to come by.

We could be in the case of having a problem of the form

$$x' = f(x, \theta), \quad x \in \mathbb{R}^d, \ t \in [0, t_{\text{max}}], \quad x(0) = x_0,$$

depending on a set of parameters $\theta in \mathbb{R}^m$. Then we have observations at discrete time points $t_1, \dots, t_p \in [0, t_{\text{max}}]$ in the form

$$(t_1, g(x^{(1)})), (t_2, g(x^{(2)})), \dots, (t_p, g(x^{(p)})),$$

for a function $g(\cdot)$ representing some observable of the system. We can state this problem again in a **least square** formulation as

$$\text{find } \theta \in \mathbb{R}^m \ : \ \min \sum_{i=1}^{p} \|g(x(t_i, \theta)) - g(x^{(i)})\|^2,$$

where $\|g(x) - g(y)\| = \sum_{i=1}^{n} |g_i(x) - g_i(y)|^2$. Thus, by manipulating a bit the quantities we have, we can use again the same strategy we have seen before.

Let us use this procedure to **estimate the parameters of a SIR model**. Let us proceed step-by-step

- first of all we need the dynamic of the system, since we will work with uncertain data, we will run this time all three differential equations

```
% We order the variables as y(t) = [S(t),I(t)]
% lambda = theta(1) gamma = theta(2)
sirModel = @(t,y,theta) [-theta(1)*y(1)*y(2); ...
    theta(1)*y(1)*y(2)-theta(2)*y(2);...
    theta(2)*y(2)];
```

- then we use the function we have seen in the last topic for integrating differential equations, this will be our workhorse, here we are using the computational resources

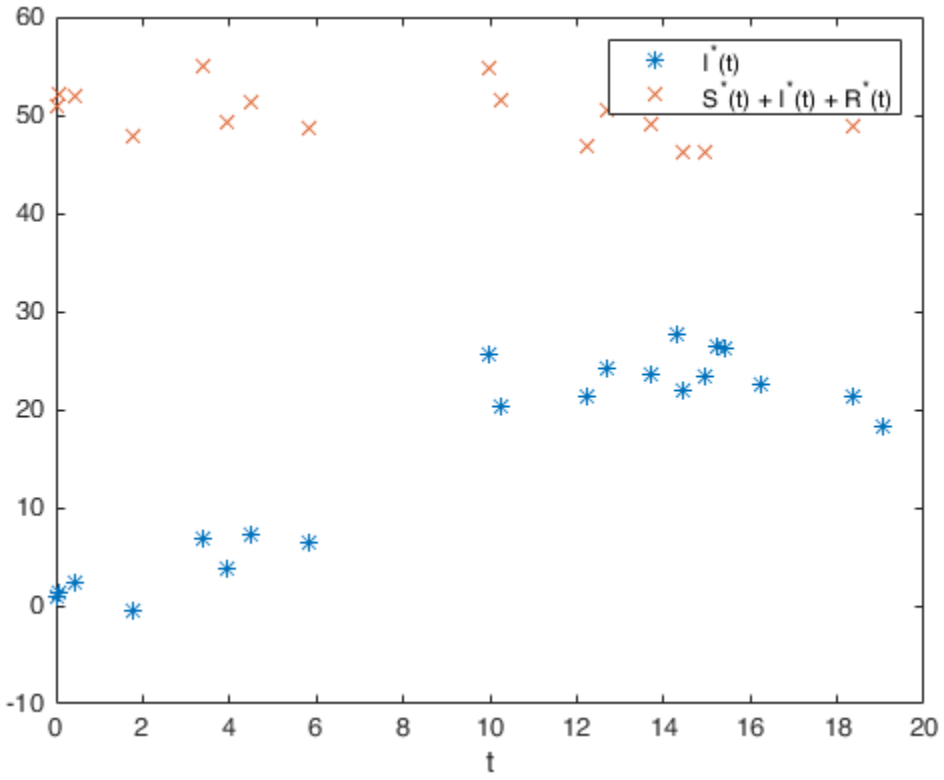
```
sirSOL = @(theta,IC,t) deval(ode45( @(t,y) sirModel(t,y,theta),t,IC),t);
```

- now we generate our noisy data, that will represent the measurements taken on the field

```
% we fix the random number generator so that we generate always the same random␣
 ↪numbers:
rng(10);
numpts=20;
tdata = [0, sort(20*rand (1,numpts))];
width = 0.1;
ndataSIR = 20*[0, normrnd(0,width,[1,numpts]);
    0, normrnd(0,width,[1,numpts]);
    0, normrnd(0,width,[1,numpts])];

lambda = 0.01;
gamma = 0.1;
S0 = 50;
I0 = 1;
R0 = 0;
theta = [lambda; gamma];
IC = [S0; I0; R0];
SIRData = sirSOL(theta,IC,tdata) + ndataSIR;
```

- now the matrix `SIRData` contains the noisy data of our system with given parameters (that are indeed the parameters that we will try to guess back). To march the optimization procedure we will focus only on the data for for $I(t)$ and $N(t)$

```
SIRDatared = [0 1 0; 1 1 1]*SIRData;

% Let us look at the data we have obtained
figure(1)
plot(tdata,SIRDatared(1,:),'*',tdata,SIRDatared(2,:),'x')
xlabel('t')
legend('I^*(t)','S^*(t) + I^*(t) + R^*(t)')
```



- We can now build the **objective** function we wish to optimize

```
SIRthetaSol = @(theta,t) [0 1 0;1 1 1]*sirSOL([theta(1) theta(2)],IC,t);
objective = @(theta) sum(sum( (SIRthetaSol(theta,tdata) - SIRDatared).^2 ));
```

- that we optimize by means of the `fmincon` function. Moreover, we tell to it to print some information on the procedure, and we request that the parameters λ and γ to be in between $[0, 0]$ and $[1, 4]$ respectively.

```
options = optimset('Display','iter');
[SIRtheta, fval, exitflag] = ...
    fmincon(objective,[0.1 4],[],[],[],[],[0 0],[1 4],[],options);
```

```
Your initial point x0 is not between bounds lb and ub; FMINCON
shifted x0 to strictly satisfy the bounds.
```

```

```

```
                                          First-order      Norm of
```

```

```

```
 Iter F-count            f(x)  Feasibility   optimality         step
```

```
    0       3    7.527718e+03    0.000e+00    3.951e+01
```

1	6	7.516116e+03	0.000e+00	1.819e+03	9.864e-01
2	9	7.516075e+03	0.000e+00	6.074e+01	8.219e-04
3	12	7.515294e+03	0.000e+00	1.149e+02	1.059e-02
4	15	7.508059e+03	0.000e+00	5.941e+02	2.065e-02
5	25	7.506838e+03	0.000e+00	1.392e+01	2.104e-02
6	28	7.503699e+03	0.000e+00	4.009e+02	9.717e-03
7	31	7.502020e+03	0.000e+00	4.160e+02	1.542e-01
8	34	6.257635e+03	0.000e+00	2.477e+05	3.827e+00
9	40	5.486048e+03	0.000e+00	5.992e+04	4.682e-01
10	50	5.364982e+03	0.000e+00	9.336e+04	1.793e-02
11	53	2.821310e+03	0.000e+00	1.527e+06	4.628e-01
12	58	1.174280e+03	0.000e+00	1.647e+04	1.735e-03

```
   13       61    1.089969e+03    0.000e+00    3.271e+05    1.275e-02

   14       64    3.673265e+02    0.000e+00    7.934e+03    6.099e-02

   15       75    2.999335e+02    0.000e+00    4.166e+03    2.358e-02

   16       78    2.935125e+02    0.000e+00    1.041e+04    5.328e-03

   17       81    2.934616e+02    0.000e+00    9.851e+03    5.052e-05

   18       84    2.932380e+02    0.000e+00    7.123e+03    2.771e-05

   19       87    2.929818e+02    0.000e+00    2.525e+02    1.504e-04

   20       90    2.929815e+02    0.000e+00    6.330e+00    1.030e-05

   21       93    2.929815e+02    0.000e+00    1.325e-02    3.758e-07

Local minimum possible. Constraints satisfied.

fmincon stopped because the size of the current step is less than
the value of the step size tolerance and constraints are
satisfied to within the value of the constraint tolerance.
```

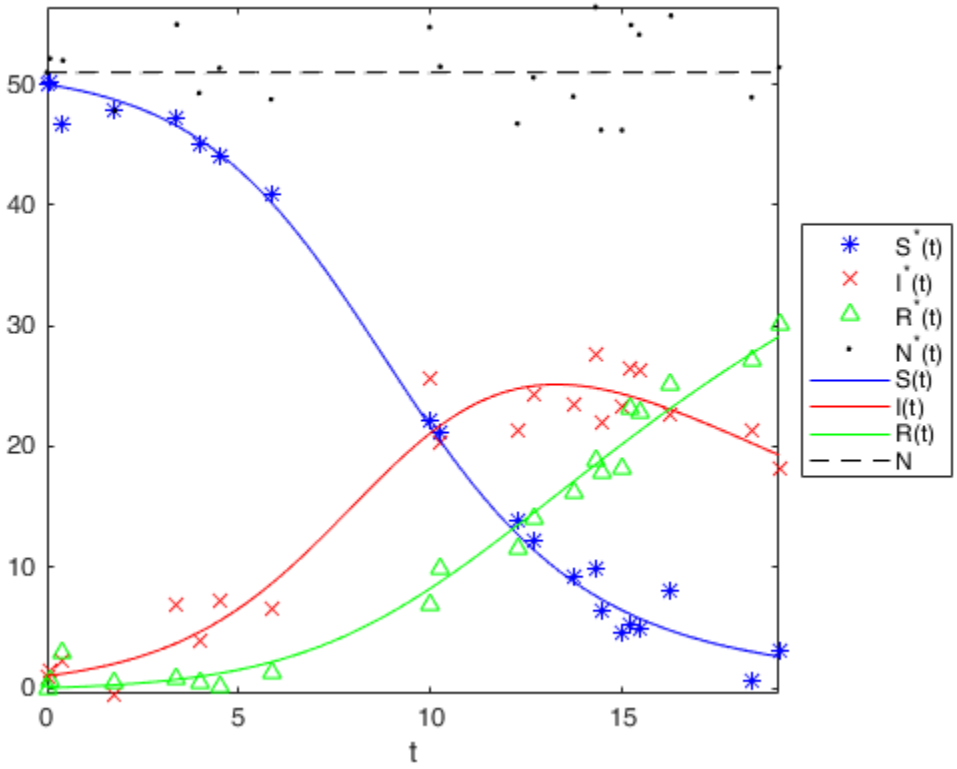
- In the final step we can print the results we have obtained to screen:

```
tsol = linspace(0,max(tdata),400);
SIRSOL = sirSOL(SIRtheta,IC,tsol);

figure(1)
plot(tdata,SIRData(1,:),'b*',...
    tdata,SIRData(2,:),'rx',...
```

(continues on next page)

(continued from previous page)

```
    tdata,SIRData(3,:),'g^',...
    tdata,sum(SIRData),'k.',...
    tsol,SIRSOL(1,:),'b-',tsol,SIRSOL(2,:),'r-',tsol,SIRSOL(3,:),'g-',...
    tsol,sum(SIRSOL),'k--');
xlabel('t')
legend({'S^*(t)','I^*(t)','R^*(t)','N^*(t)',...
    'S(t)','I(t)','R(t)','N'},'Location','eastoutside')
axis tight
```



CHAPTER

FIVE

GRAPHS AND NETWORKS

Understanding complex systems often requires a bottom-up analysis. This can be done by examining the elementary parts of the system individually and then by turning the analysis towards the connection between them. A natural way of performing this type of analysis is through the use of **networks** or **graphs**.

Definition 1

A **graph** $G = (V, E)$ consists of a finite set V of **vertices** (or nodes) and a finite set E of **edges**, where each edge $e \in E$ is of the form $e = \{u, v\}$ with $u, v \in V$.

And we will make here the following simplifying assumptions:

- we do not allow any edge to be a self-loop, i.e., an edge that starts and ends at the same vertex,
- we do not allow more than one edge between any pair of vertices,
- unless mentioned otherwise we will mostly consider undirected edges, that is edge $e = \{u, v\} = \{v, u\}$.

Now, let us fix some notation and nomenclature.

Definition 2

Two vertices, u and v, are called **adjacent** if there is an edge, uv, connecting them. We can express adjacency of u and v by writing $u \sim v$.

Definition 3

If v is a vertex in a graph G, the **degree** of v, denoted $\deg(v)$, is the number of edges adjacent to v.

Definition 4

A **walk** in a graph is a sequence of vertices and edges v_0, e_1, v_1, e_2, v_2, ..., v_{k-1}, e_k, v_k such that edge e_i connects vertices v_{i-1} and v_i, for $1 \leq i \leq k$.

Definition 5

A graph G is **connected** if for any two vertices u and v of G, there is a *walk* in G from u to v.

The first way to store a graph into a computer is as an **adjacency matrix**. Given a graph G with n vertices, we start by label them from 1 to n, then the adjacency matrix representing this graph will be an $n \times n$ matrix whose entries are either

0 or 1, specifically

$$A(i,j) = \begin{cases} 1, & (v_i, v_j) \in E, \\ 0, & \text{otherwise.} \end{cases}$$

We now have much of the dictionary we need to start investigating some examples of graphs and networks that are used in applications. We will uncover some more information by working through the test cases.

5.1 A social network made of dolphins

How can we use these abstract objects to model some systems of interest? We have said from the beginning that graphs are useful to describe pairwise interactions between vertices. We will start as usual from an example, and in particular we will focus on the network of social relationship between a population of Doubtful Sound bottlenose dolphins [Lus03]. It has been observed that gregarious, long-lived animals, such as gorillas (*Gorilla gorilla*), deer (*Cervus elaphus*), elephants (*Loxodonta africanus*) and bottlenose dolphins (*Tursiops truncatus*) rely on information transfer to exploit their habitat. Thus investigating their social structure is surely of interest.

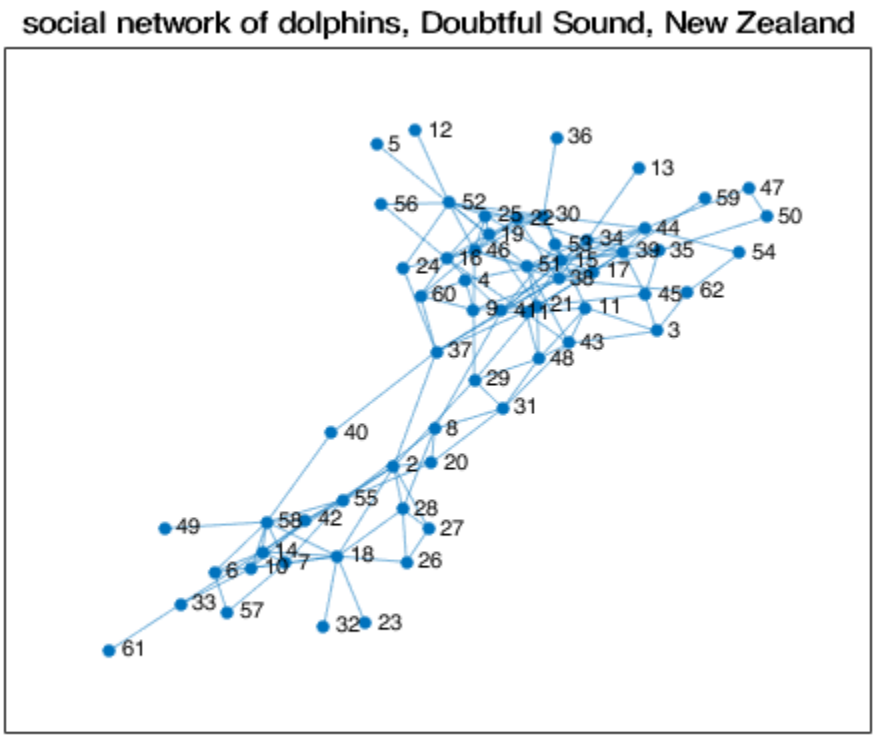
First of all, let us fetch and load the data into the MATLAB environment

```
websave('dolphins.mat','https://suitesparse-collection-website.herokuapp.com/mat/
 ↪Newman/dolphins.mat');
load('dolphins.mat')
disp(Problem)
```

```
      name: 'Newman/dolphins'
     title: 'social network of dolphins, Doubtful Sound, New Zealand'
         A: [62x62 double]
        id: 2396
      date: '2003'
    author: 'D. Lusseau'
      kind: 'undirected graph'
     notes: [19x75 char]
       aux: [1x1 struct]
        ed: 'M. Newman'
```

The variable `Problem` is a `struct` variables containing several information, what mostly concerns us now is the field `Problem.A` that contains the **adjacency** matrix of our graph. We can now use MATLAB to transform it into the graph format and visualize it

```
G = graph(Problem.A);
figure(1)
plot(G)
title(Problem.title)
```

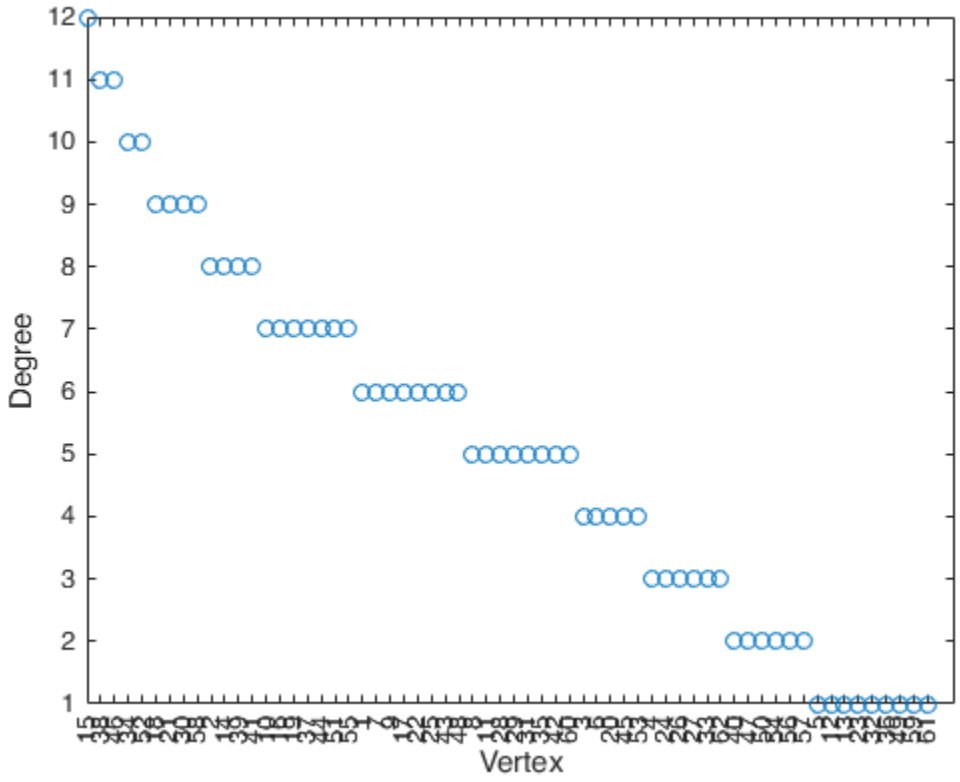


What are we seeing? In the word of the *author* [Lus03]:

> Every time that a school of dolphins was encountered in the fjord between 1995 and 2001, each adult member of the school was photographed and identified from natural markings on the dorsal fin. This information was used to determine how often two individuals were seen together. To measure how closely two individuals were associated in the population (i.e. how often they were to be found together) I calculated a half-weight index (HWI) of association for each pair of individuals (Cairns & Schwäger 1987). This index estimates the likelihood that two individuals would be seen together compared with the likelihood of seeing any of the two individuals when encountering a school: [...] Over the 7 years of observation the composition of 1292 schools was gathered. There were 64 adult individuals in this social network linked by 159 preferred companionships (edges)

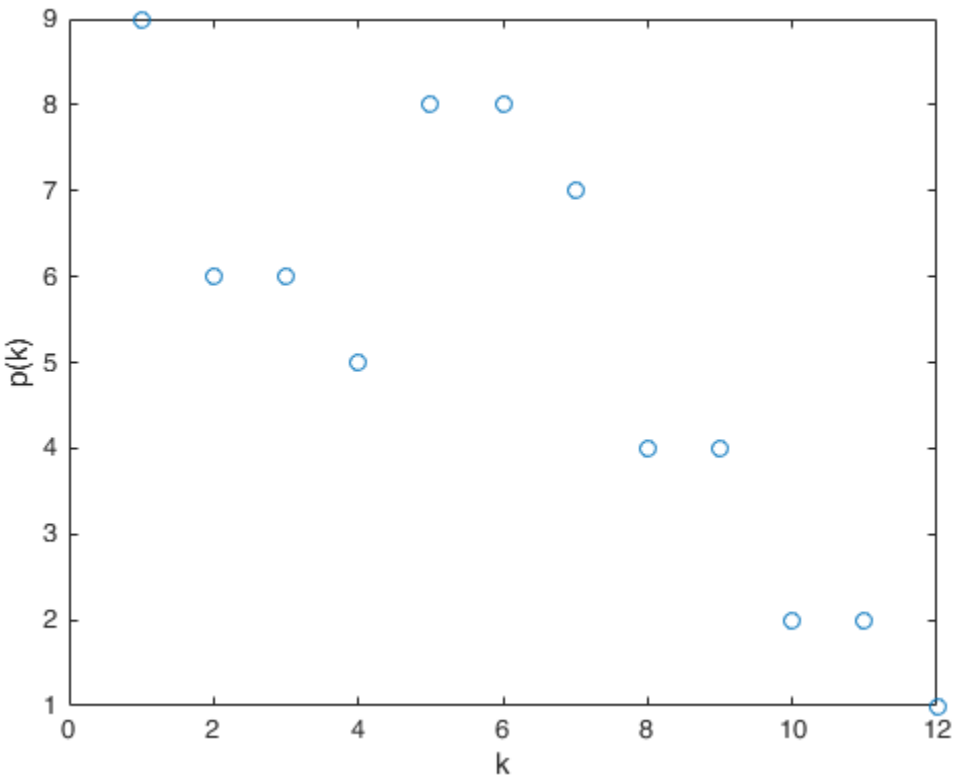
Now let us try to investigate some properties of the network to infer some information on the dolphins. The first thing we want to look at are the **degrees** of the vertexes:

```
degree = G.degree;
[sorted_degree,rank] = sort(degree,'descend');
figure(2)
plot(1:G.numnodes,sorted_degree,'o');
axis([1 64 1 max(degree)])
xlabel('Vertex')
ylabel('Degree')
xticks(1:G.numnodes)
xticklabels(rank)
xtickangle(90)
```



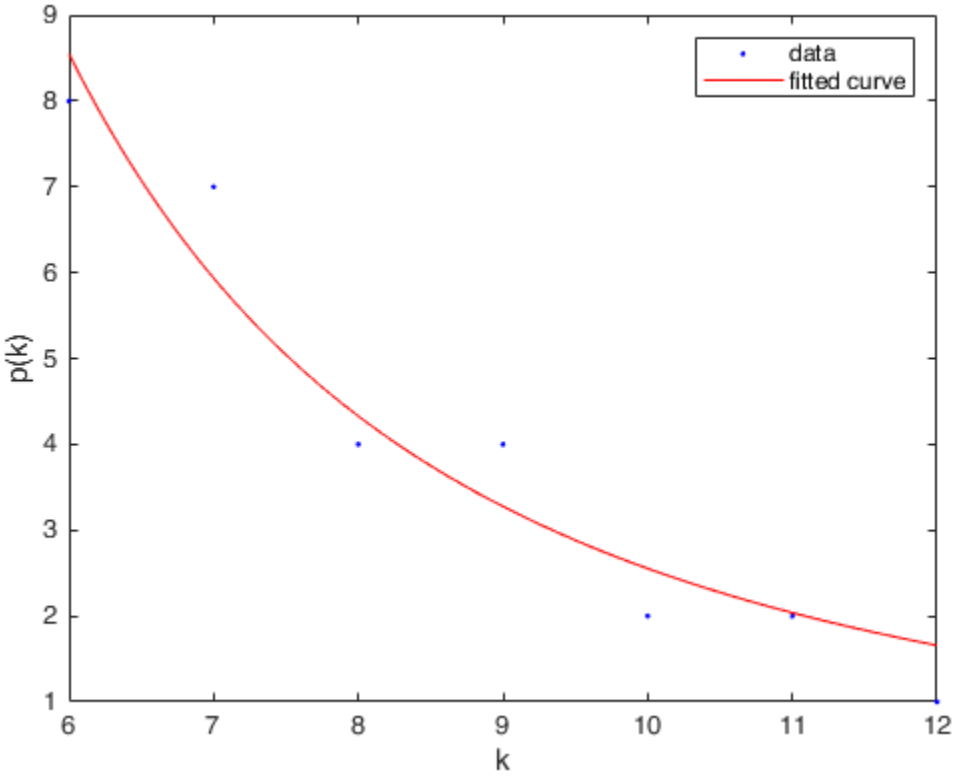
The behavior of the (reordered) degree distribution seem suspicious… let us look more into it, we call k the degree and $p(k)$ the number of nodes with that degree

```
[gc,gr] = groupcounts(degree);
figure(3)
plot(gr,gc,'o')
xlabel('k')
ylabel('p(k)')
```



Now we observe that the behavior for smaller degree is uncertain, while the data clearly exhibit a *tail* (an *asymptotic*) that decays as a power law. Let us use the techinique for parameter estimation we have seen in the last topic for this case

```
fo = fitoptions('Method','NonlinearLeastSquares',...
                'Lower',[0 0],...
                'Upper',[Inf Inf],...
                'StartPoint',[1 1]);
ft = fittype('a*x.^(-b)','options',fo);
[curve,gof] = fit(gr(6:end),gc(6:end),ft);
% We plot the results
figure(1)
plot(curve,gr(6:end),gc(6:end))
xlabel('k');
ylabel('p(k)');
```



We have obtained a reasonable fit

```
disp(curve);
```

```
     General model:
     curve(x) = a*x.^(-b)
```

```
     Coefficients (with 95% confidence bounds):
       a =       592.1  (-381.8, 1566)
       b =       2.365  (1.523, 3.208)
```

with $p(k) \sim k^{-2.365}$.

Scale-free networks

A **scale-free** network is a network whose degree distribution follows a power law, at least *asymptotically*. That is, the fraction $p(k)$ of nodes in the network having degree k behaves for large values of k as

$$p(k) \sim k^{-\alpha}, \quad 2 < \alpha < 3.$$

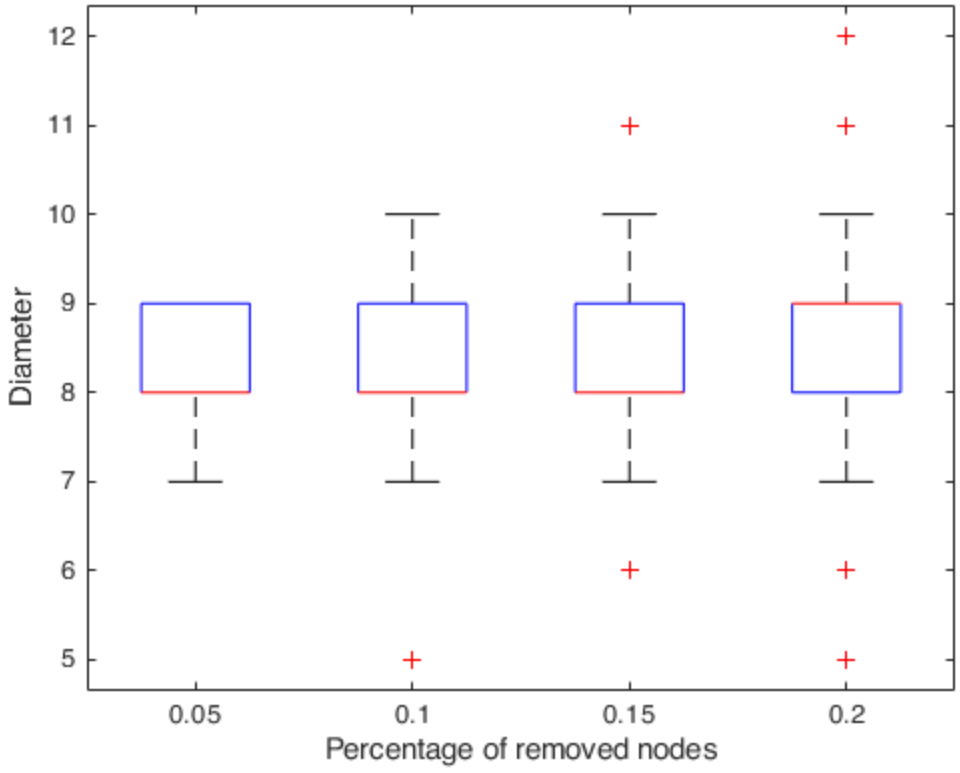
This type of networks have usually several properties that we can use to interpret the underlying phenomena. One of them is that they usually have a *small diameter*, that is, the length of the "longest shortest path", i.e., the largest number of vertices which must be traversed in order to travel from one vertex to another when paths which backtrack, detour, or loop are excluded from consideration. We can compute it by doing

```
diameter = max(distances(G),[],'all');
disp(diameter)
```

```
     8
```

Thank to this property, dolphin scale-free network is *resilient to random attacks*. If we remove some random nodes the increase in the diameter is small:

```
percentage_of_removed = [0.05,0.10,0.15,0.20];

for j=1:length(percentage_of_removed)
    percentage = percentage_of_removed(j);
    for i = 1:500
        H = G;
        removednodes = randi(G.numnodes,floor(percentage*G.numnodes),1);
        H = H.rmnode(removednodes);
        [bin,binsize] = conncomp(H);
        idx = binsize(bin) == max(binsize);
        SG = subgraph(H, idx);
        diameters(j,i) = max(distances(SG),[],'all');
    end
end

figure(4)
boxplot(diameters.')
ylabel('Diameter')
xlabel('Percentage of removed nodes')
xticklabels(percentage_of_removed);
```



5.2 Finding communities

Let us start again from an example, we consider here some data from [vDKArguellesTico+14] related to the behavior of a specie of *social birds* that nest in communal chambers. These contains a set of networks constructed in the following way:

> An individual was assigned to a given nest chamber once it had been observed to enter it, irrespective of the activity carried out, i.e. either building the nest chamber or roosting in it. A network 'edge' was drawn between individuals that used the same nest chambers either for roosting or nest-building at any given time within a series of observations at the same colony in the same year, either together in the nest chamber at the same time or at different times. These individuals werethus assumed to be associated.

We read from the data file the nodes, and edges of the network. The file (that we obtained from the Network Repository [RA15]) is not formatted as a CSV file, but has instead spaces to separate the data, thus we use the command `dlmread`, that generalizes the `csvread` command

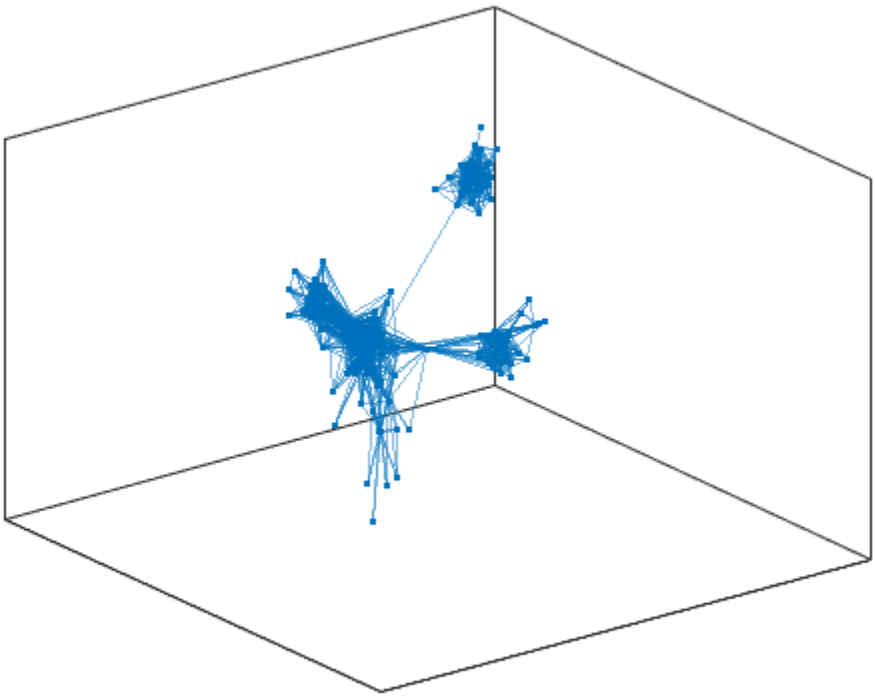
```
data = dlmread('aves-wildbird-network-1.edges');
```

with this we have obtained a matrix with three columns and number of edges rows in which the first column represents the starting node, the second column the ending node, and the third one the edge weight. With this data we can build a graph

```
G = graph(data(:,1),data(:,2),data(:,3));
```

and then plot it

```
plot(G,'Layout',"force3");
```



From the plot we suspect that we can identify two communities in our set of birds, but how can we do it mathematically? This task is called a task of **community detection**, and there are many algorithms and strategies for achieving it. We are going to focus here on a techinique that is called **spectral clustering**. First of all we need a particular representation of the network with a *matrix* called a *Laplacian of the network*:

```
L = laplacian(G);
```

Then we recover the **spectral information** by the command:

```
[v,l] = eigs(L,2,'smallestabs');
```

This gives us two vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ and two values λ_1 and λ_2 such that

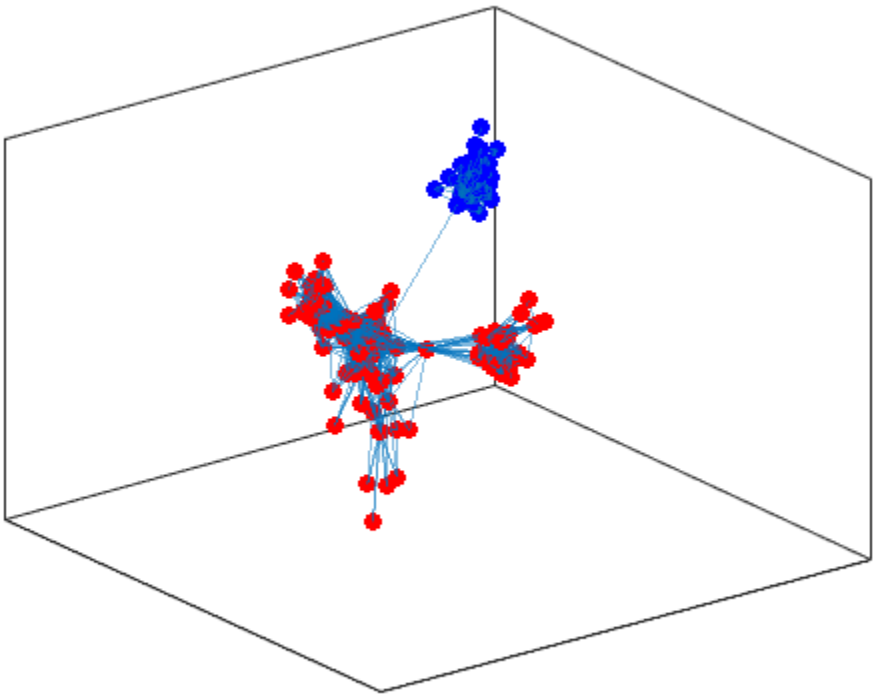
$$L\mathbf{v}_1 = \lambda_1\mathbf{v}_1, \quad L\mathbf{v}_2 = \lambda_2\mathbf{v}_2,$$

these are called, respectively, **eigenvectors**, and **eigenvalues**. If we inspect them we observe that $v_1(i) > 0$ for all $i = 1, \dots, N$, and moreover it has a constant value. The second one, is the one we are actually interested in, this has both values that are > 0 and < 0. We are going to use them to determine the communities:

```
ind1 = find(v(:,2) > 0);
ind2 = find(v(:,2) < 0);
```

Now let us evidentiate the nodes on the graph

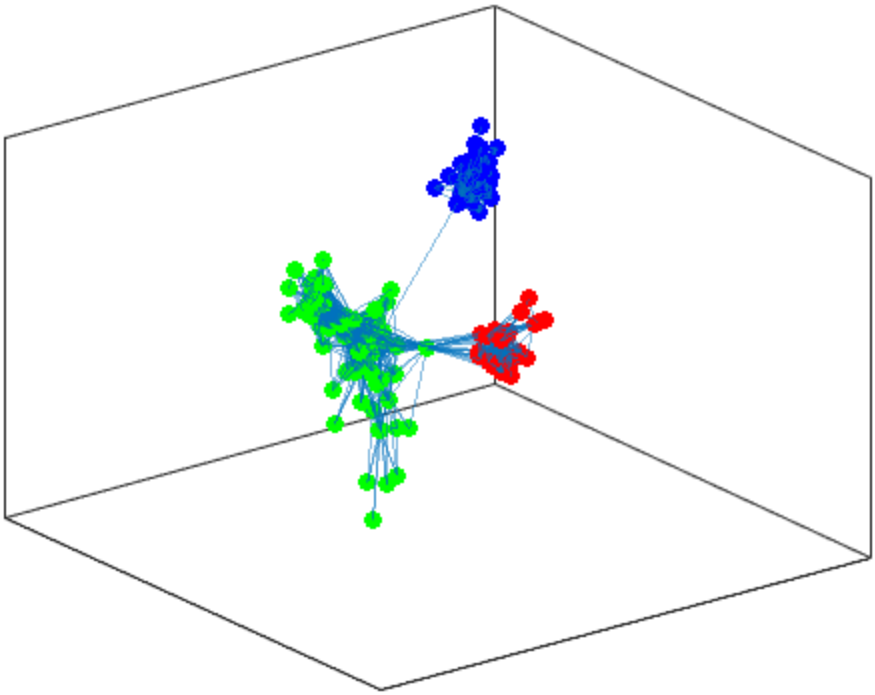
```
h = plot(G,'Layout',"force3");
highlight(h,ind1,"NodeColor",'red',"MarkerSize",6)
highlight(h,ind2,"NodeColor",'blue',"MarkerSize",6)
```



and as you can observe the procedure did evidentiate the two communities we were suspecting by working directly on the data. On the other hand, now that we look better at the data, we suspect that there could be more than two communities,

the larger one seems to be splittable in two. We can try to identify larger number of communities, by **computing more eigenvectors**

```
[v,l] = eigs(L,3,'smallestabs');
idx = kmeans(v(:,2:3),3);
ind1 = find(idx == 1);
ind2 = find(idx == 2);
ind3 = find(idx == 3);
h = plot(G,'Layout',"force3");
highlight(h,ind1,"NodeColor",'red',"MarkerSize",6)
highlight(h,ind2,"NodeColor",'blue',"MarkerSize",6)
highlight(h,ind3,"NodeColor",'green',"MarkerSize",6)
```



now we don't have a clear cut between positive and negative values in a single vector, thus we have to employ another algorithm to do the separation for us. This is the K**-means** algorithm that returns as a vector of indexes for the corresponding communities.

Tip: K-means is a *clustering method* that uses the process of **vector quantization**, it was originally devised in the signal processing community with the aims of partitioning n observations into k clusters in which each observation belongs to the cluster with the nearest mean, i.e., the mean servs as a prototype of the cluster.

It can be formally expressed as the minimization problem

$$\arg \min_{\mathbf{S}=\{S_1,\dots,S_k\}} \sum_{i=1}^{k} \sum_{\mathbf{x}\in S_i} \|\mathbf{x} - \boldsymbol{\mu}_i\|^2,$$

where we are given $\{\mathbf{x}_1, \dots, \mathbf{x}_n\}$ observations, to cluster in $k\,(\leq n)$ groups $\{S_1, \dots, S_k\}$, and $\boldsymbol{\mu}_i$ is the mean of the points

in S_i.

But how do we know if we haven't lost some community in our analysis? To test this idea we can use another concept from graph theory that is called **modularity**.

Definition 6

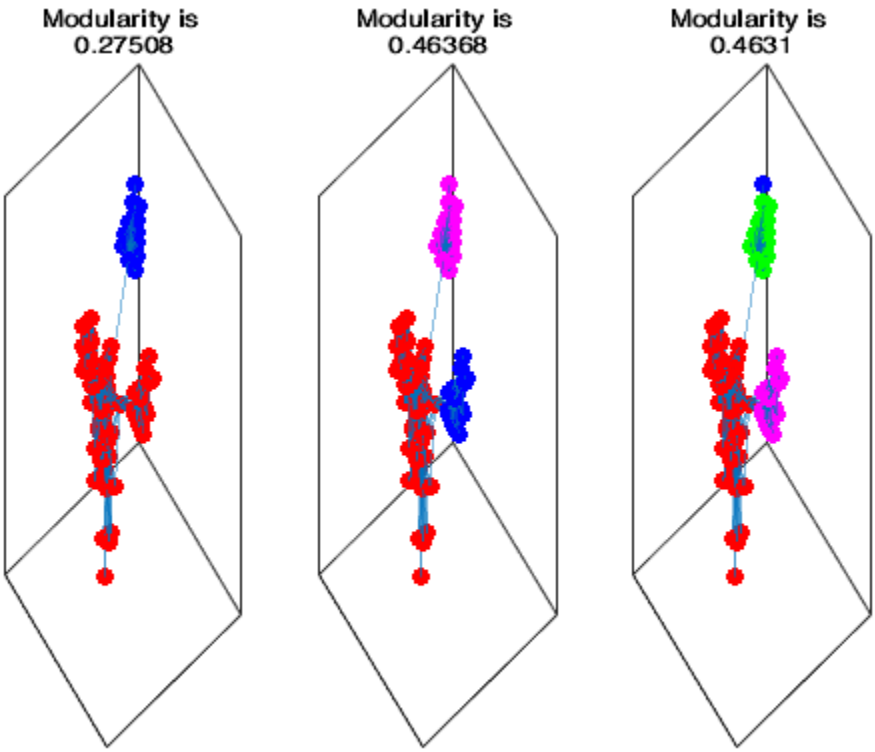
Modularity is a measure of the structure of networks or graphs which measures the strength of division of a network into communities, it is computed as the fraction of the edges that fall within the given groups minus the expected fraction if edges were distributed at random.

There are several ways for computing this quantity, we give here a very straigthforward

```
function Q = modularity(A, g)
%% MODULARITY computes the modularity of the of the partition in group g of the
% graph with adjacency matrix A.

nCommunities = numel(unique(g));
nNodes = length(g);

e = zeros(nCommunities);
for i = 1:nNodes
    for j = 1:nNodes
        e(g(i), g(j)) = e(g(i), g(j)) + A(i, j);
    end
end
nEdges = sum(A(:));  % we could use nnz(A), but we want to take into account weights
a_out = sum(e, 2);   % out-degree
a_in = (sum(e, 1))'; % in-degree
a = a_in.*a_out/nEdges^2;
Q = trace(e)/nEdges - sum(a);

end
```

and we try to use it to evaluate the communities we have found

```
%% Analysis of Community Structure
addpath('matlabcodes')
data = dlmread('aves-wildbird-network-1.edges');
color = {'red','blue','magenta','green'};
G = graph(data(:,1),data(:,2),data(:,3));
L = laplacian(G);
for i = 1:3
    [v,l] = eigs(L,i+1,'smallestabs');
    groupvec = kmeans(v(:,2:i+1),i+1);
    figure(1)
    subplot(1,3,i);
    h = plot(G,'Layout',"force3");
    for j = 1:i+1
       ind = find(groupvec == j);
       highlight(h,ind,"NodeColor",color{j},"MarkerSize",6)
    end
    Q = modularity(adjacency(G),groupvec);
    title(['Modularity is ',string(Q)])
end
set(gcf,'Position',[-1984 426 1301 395]);
```



from which we observe that if we try to find more than three communities the modularity starts decreasing. Thus it seems that we are done with three communities for this dataset.

Exercise 8

We write a function that looks for in a graph up to n communities and returns the number of communities within n with the highest modularity.

```
function [groups,outmodularity,optimum] = findcommunities(G,k)
%%FINDCOMMUNITIES looks in G up to n communities and returns the number
%of communities within n with the highest modularity.
%   G = graph
%   k = maximum number of communities

if (k > G.numnodes)
    error("You cannot have more than number of nodes communities");
end
if (k < 2)
    error("You cannot have less than 2 communities");
end

% Allocate the space for the groups matrix of number of nodes rows and
% k-1 columns

% Allocate the space for the modularities: vector with k-1 entries

% Loop through the possible community size
% | Compute eigenvectors
```

(continues on next page)

(continued from previous page)

```
% | Use k-means
% | Compute modularity
% end of the for loop

% Compute the index for which maximum modularity is obtained

end
```

You can test the algorithm on some of the networks on some Animal social networks.

Tip: To ensure that the function also automatically produces graphs highlighting the communities, the following two auxiliary codes can be used:

- Generate maximally perceptually-distinct colors This function generates a set of colors which are distinguishable by reference to the "Lab" color space, which more closely matches human color perception than RGB.
- neatly arrange subplots Sometimes a graphing function will not know in advance how many sub-plots are to be created. This function produces reasonable values for the row and column inputs to subplot given the number of desired sub-plots.

5.3 Centrality

The next type of analysis we want to address is the computation of a **node centrality score**, this type of analysis can (usually) be done by employing some *topological* measures, i.e., that depends only on the connections of the nodes, to score nodes by their "importance".

Definition 7

A **graph Centrality measures** are scalar values given to each node in the graph $G = (V, E)$ to quantify its *importance*. The definition of what is *important* depends on the underlying model assumption.

Let us see some **measures** on a sample graph $G = (V, E)$ that could be, e.g., the *contact network* ia-infect-hyper.mtx where nodes represent humans and edges between them represent proximity, i.e., a contact for a given period of time in the physical world; see [RA15].

```
addpath('./data')
data = dlmread('ia-infect-hyper.mtx',' ',2,0);
G = graph(data(:,1),data(:,2));
fprintf("G is a Network with %d nodes and %d edges.\n",G.numnodes,G.numedges);
```

```
G is a Network with 113 nodes and 2196 edges.
```

- **Degree centrality**. It is defined as the number of node neighbors for each node in the graph. If the network is directed, we have two versions of this measure: the *in-degree* is the number of incoming edges, and the *out-degree* that is in turn the number of out-going edges. This is a *local measure*, that means that as a measure it doesn't take neighbors connectivity into account. It can be interpreted as a form of popularity.

```
degree_centrality = G.degree;
```

- **Closeness centrality**. This centrality measures node efficiency in terms of connection to other nodes. Is defined as the average length of the shortest path between the node and all other nodes in the graph, i.e., the more central a node is, the closer it is to all other nodes.

$$v \in V, \quad C(v) = \frac{1}{\sum_{w \in V} d(v,w)}.$$

```
closeness_centrality = centrality(G,"closeness");
```

- **Betweenness centrality**. This measure quantifies the number of times a node acts as a *bridge* along the shortest path between two other nodes, that is, let us say that we want to compute it for a vertex $v \in V$. We first compute **all** the shortest paths between each pair of vertices (s,t), then for each of the couples we determine the fraction of shortest paths that pass through v. The measure is then the sum this fraction over all pairs of vertices, succinctly

$$v \in V, \quad B(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}, \quad \begin{array}{l} \sigma_{st} = |\{\text{shortest paths between s and t }\}|, \\ \sigma_{st}(v) = |\{\text{shortest paths between s and t passing per v}\}|. \end{array}$$

```
betweenness_centrality = centrality(G,"betweenness");
```

- **Eigenvector centrality**. It assigns relative scores to all nodes in the network based on the *assumption* that "connections to important nodes matters more to the score of the node we are looking at than equal connections to low importance nodes". This idea can be formalized in different ways. The first we consider is the case in which we use the *eigenvector* corresponding to the *largest eigenvalue* of the graph adjacency matrix.

```
eigenvector_centrality = centrality(G,"eigenvector");
```

- **PageRank**. This is the algorithm used by Google Search to rank the web pages in their search engine results. The score given by this algorithm is a *probability distribution* representing the likelihood of randomly exploring the network and arriving at any particular page.

$$\mathbf{p} = \mathbf{p}\left(\gamma P + (1-\gamma)\mathbf{v}\mathbf{1}^T\right), \quad P = \text{diag}(A\mathbf{1})^{-1}A, \ \gamma \in [0,1], \ \mathbf{1}^T\mathbf{v} = 1.$$

```
pagerank_centrality = centrality(G,"pagerank");
```

We can now try to look at the most-important nodes for the different measures. Since we are only interested in the ranking (and not in the actual value of the measure) we will make use of the `sort` function from MATLAB to get the information we want

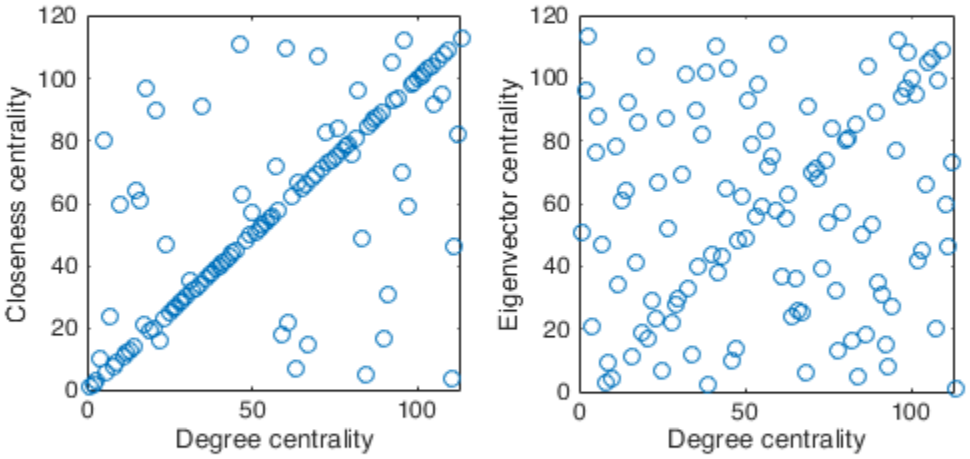
```
[~,degree_rank] = sort(degree_centrality,"descend");
[~,closeness_rank] = sort(closeness_centrality,"descend");
[~,betweenness_rank] = sort(betweenness_centrality,"descend");
[~,eigenvector_rank] = sort(eigenvector_centrality,"descend");
[~,pagerank_rank] = sort(pagerank_centrality,"descend");

rankings = table(degree_rank,closeness_rank,betweenness_rank,...
    eigenvector_rank,pagerank_rank,'VariableNames',...
    {'Degree','Closeness','Betweenness','Eigenvector','PageRank'});
disp(head(rankings))
```

```
    Degree    Closeness    Betweenness    Eigenvector    PageRank
    ______    _________    ___________    ___________    ________

      30          30            30             30           30
```

```
    38          38          42         102          42

    42          42          33          38          38

   102         102          38          42         102

    33          33         102          33          33

    48          48          12          48          48

    12          12          48          34          12
    34          34          34          12          34

```

To compare the different rankings we can also look at the **scatter plot** of the rankings, e.g.,

```
figure(1)
subplot(1,2,1);
plot(degree_rank,closeness_rank,'o');
xlabel('Degree centrality')
ylabel('Closeness centrality')
axis square
subplot(1,2,2);
plot(degree_rank,eigenvector_rank,'o');
xlabel('Degree centrality')
ylabel('Eigenvector centrality')
axis square
```



In general it would be better to have also a **quantitative way** of comparing the rankings. We can go for a *statistical test*, in this case a good solution is the Kendall τ test.

The Kendall τ coefficient is defined as:

$$\tau = \frac{(\text{number of concordant pairs}) - (\text{number of discordant pairs})}{\binom{n}{2}}.$$

Where $\binom{n}{2} = \frac{n(n-1)}{2}$ is the **binomial coefficient** for the number of ways to choose two items from n items. We can compute it in MATLAB (together with the relevant p-value) by doing:

```
[tau,pval] = corr(degree_rank,closeness_rank,'type','Kendall');
fprintf("Degree vs Closeness tau = %f p-value = %e.\n",tau,pval);
[tau,pval] = corr(degree_rank,eigenvector_rank,'type','Kendall');
fprintf("Degree vs Eigenvector tau = %f p-value = %e.\n",tau,pval);
```

```
Degree vs Closeness tau = 0.581226 p-value = 7.253673e-20.
```

```
Degree vs Eigenvector tau = 0.099558 p-value = 1.185833e-01.
```

Warning: Every **centrality measure** is based on an assumption about the concept of importance, e.g, *closeness* and *betweenness* centralities define the importance as an evaluation of the information exchange efficiency within the networks. Others like the degree are purely local popularity measure, while the ones based on the eigenvectors tend to reward highly connected nodes. You should always select your measure while taking into account the model behind the choice.

CHAPTER

SIX

WORKING WITH IMAGES

Up to now we have worked with numerical data (data from experiments to be adapted to models, simulations of differential equations) and relational data (networks). This isn't the only type of data you can happen to be dealing with. Another rather frequent occurrence is that of having to analyze images from various sources. We will deal here with the solution of some problems that concern them.

The **first issue** we have to solve, is indeed interfacing the computer with this type of data. MATLAB commands can *read*, *write*, and *display* several types of image file formats:

- BMP
- GIF
- HDF
- JPEG
- PCX
- PNG
- TIFF
- XWD

Let us start by downloading from the internet a cat photo and loading it into MATLAB:

```
websave('cat.jpg','https://upload.wikimedia.org/wikipedia/commons/5/52/Panthera_leo_
 ↪stretching_%28Etosha%2C_2012%29.jpg');
A = imread("cat.jpg");
```

As we have seen many times, the fundamental data in MATLAB is the *array*, and thus the variable `A` we have obtained with the `imread` command is indeed a type of array. In many cases, images can be represented by using matrices, in which entry corresponds to a *single pixel* of the given image. Some images, for example RGB color images, require instead a tensor to be represented, that is, if you prefer, a three-dimensional array. In the first plane (with respect to the third dimension) we represent the red pixel intensities, in the second one we represent the green pixel intensities, and finally in the third one the blue pixel intensities.

Indeed, if we query for the variable we have just created, we find as much

```
whos A
```

```
  Name          Size                    Bytes  Class    Attributes

  A          2000x3000x3             18000000  uint8
```

that is A is a vector of integers of sizes $2000 \times 3000 \times 3$. We can look at the figure it represents by doing

```
imshow(A);
```



Other general utility functions that we can consider are

- converting an RGB image to grayscale

```
Agray = rgb2gray(A);
imshow(Agray);
```



- create a *tiled image* from multiple images

```
tile = imtile({A,Agray});
imshow(tile);
```



- cropping and saving a reduced size image

```
image(A); axis image;
p = ginput(2);
% Get the x and y corner coordinates as integers
sp(1) = min(floor(p(1)), floor(p(2))); %xmin
sp(2) = min(floor(p(3)), floor(p(4))); %ymin
sp(3) = max(ceil(p(1)), ceil(p(2)));   %xmax
sp(4) = max(ceil(p(3)), ceil(p(4)));   %ymax
CroppedA = A(sp(2):sp(4), sp(1): sp(3),:);
figure; image(CroppedA); axis image
imwrite(CroppedA,'cropped_cat.jpg');
```



in which we have used the `ginput` command to get the coordinates of some points from the figure, and the `imwrite` command to save the cropped figure to a file.

- obtaining information about graphics files (maybe useful before loading them)

```
iminfo = imfinfo('cat.jpg')
```

```
Warning: Division by zero when processing .  The value has been set to NaN.
> In matlab.io.internal.imagesci.imjpginfo>incorporate_exif_metadata (line 68)
In matlab.io.internal.imagesci.imjpginfo (line 51)
In imjpginfo (line 20)
In imfinfo (line 234)
```

```
iminfo =

  struct with fields:

          Filename: '/home/cirdan/Documenti/RTDa-PISA/Didattica/NumRecipes/
↪recipesforenvsciences/src/cat.jpg'
       FileModDate: '23-Mar-2022 23:20:15'
          FileSize: 5210752
            Format: 'jpg'
     FormatVersion: ''
             Width: 3000
            Height: 2000
          BitDepth: 24
         ColorType: 'truecolor'
   FormatSignature: ''
   NumberOfSamples: 3
      CodingMethod: 'Huffman'
     CodingProcess: 'Sequential'
           Comment: {}
              Make: 'Canon'
```

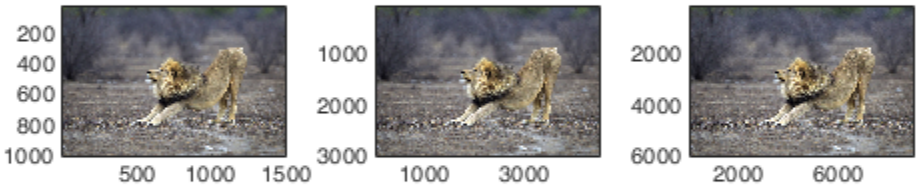
(continues on next page)

(continued from previous page)

```
              Model: 'Canon EOS-1D Mark IV'
        Orientation: 1
        XResolution: 240
        YResolution: 240
     ResolutionUnit: 'Inch'
           Software: 'Adobe Photoshop CS2 Windows'
           DateTime: '2014:01:26 13:25:17'
             Artist: 'Yathin Krishnappa'
          Copyright: 'yathin.com'
      DigitalCamera: [1x1 struct]
      ExifThumbnail: [1x1 struct]
```

- resizing images, if you want to enlarge a figure you need to *guess* somehow the value of the missing pixels, similarly, if you reduce it, you have to *decide* how to modify the remaining ones. This is done via an *interpolation kernel* function that calculates the value of a pixel using a weighted average of neighboring pixel values

```
subplot(1,3,1)
Ares = imresize(A, 0.5, "Method","bicubic");
image(Ares); axis image;
subplot(1,3,2)
Ares = imresize(A, 1.5, "Method","bilinear");
image(Ares); axis image;
subplot(1,3,3)
Ares = imresize(A, 3, "Method","nearest");
image(Ares); axis image;
```

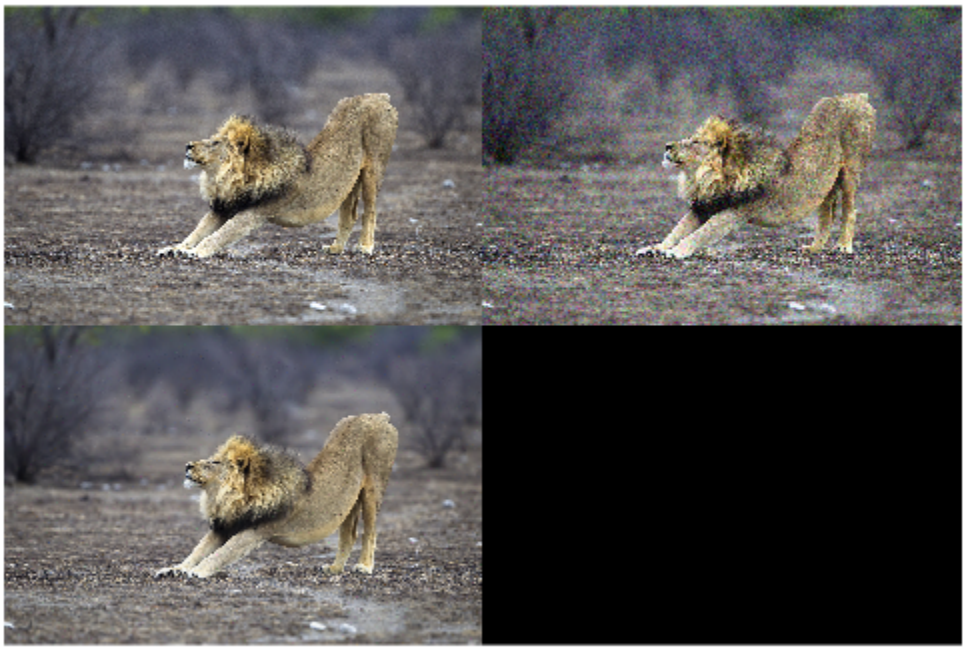


6.1 Denoising images

A part from photos of cats sometimes we get photos from experimental instruments that are littered with noise, and that we wish to remove. This may be either for simply having a better looking photo, or as a first step in a following analysis.

From the **wavelet Toolbox** you can use the following procedure to *denoise* an image:

```
Anoise = imnoise(A,'gaussian',0,0.01);
tile = imtile({A,Anoise});
imshow(tile);
Adenoise = wdenoise2(Anoise);
tile = imtile({A,Anoise,uint8(Adenoise)});
imshow(tile);
```



Tip: A **wavelet transforms** of a given signal of finite energy acts as a projection on a continuous family of *frequency bands*. We can represent, for instance, the signal could be represented on every frequency band $[f, 2f]$ for all $f > 0$, for example for $f = 1$ one could select the function

$$\phi(t) = \frac{\sin(2\pi t) - \sin(\pi t)}{\pi t},$$

and for the bands $[1/a, 2/a]$ by the function

$$\phi_{a,b} = \frac{1}{\sqrt{a}}\phi\left(\frac{t-b}{a}\right), \quad b \in \mathbb{R}.$$

Then a signal $x(t)$ can be decomposed as

$$x_a(t) = \int_{\mathbb{R}} W_\phi\{x\}(a,b) \cdot \phi_{a,b}(t)\,\mathrm{d}b, \quad W_\phi\{x\}(a,b) = \int_{\mathbb{R}} x(t)\phi_{a,b}(t)\,\mathrm{d}t.$$

Since it is *computationally intractable* to analyze a continuous signal this way, we usually reduce to a *smaller set of discrete coefficients*

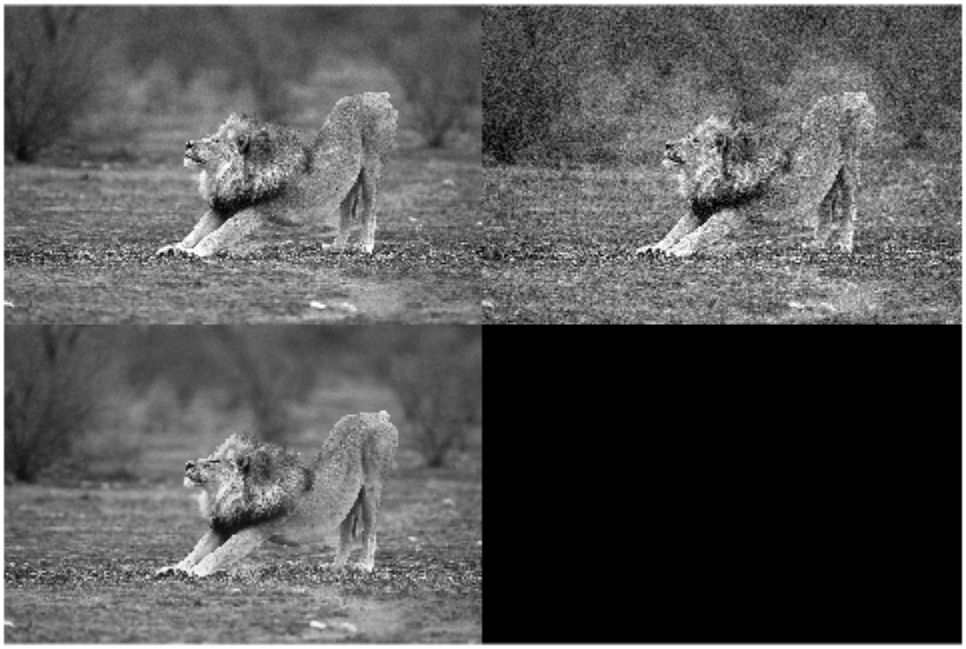
$$x(t) = \sum_{m\in\mathbb{Z}} \sum_{n\in mathbbZ} C_{m,n}\phi_{m,n}(t), \quad \phi_{m,n}(t) = \frac{1}{\sqrt{a^m}}\phi\left(\frac{t-nb}{a^m}\right), \quad C_{m,n} = \int_{\mathbb{R}} x(t)\phi_{m,n}(t)\,\mathrm{d}t.$$

In any case, when we have computed the coefficient, what we do to *denoise the image* is to throw away the "small one" that are mostly noises, while the "large ones" contain actual signal.

Further options can be analyzed by looking at the `help wdenoise2`.

This is a classic way of solving this kind of problem. In recent times, another widely used approach is that of **neural networks**. The **Image Processing MATLAB Toolbox** has a *pretrained* neural network for denoising *black and white* images, it can be called by doing:

```
Agray = rgb2gray(A);
Agraynoise = imnoise(Agray,'gaussian',0,0.01);
net = denoisingNetwork('DnCNN');
Adenoisenet = denoiseImage(Agraynoise,net);
tile = imtile({Agray,Agraynoise,uint8(Adenoisenet)});
imshow(tile);
```



the network `denoisingNetwork('DnCNN')` is a rather complex object whose structure can be investigated by doing

```
disp(net.Layers)
```

```
  59x1 Layer array with layers:
```

```
    1   'InputLayer'              Image Input           50x50x1 images

    2   'Conv1'                   Convolution           64 3x3x1 convolutions with␣
↪stride [1  1] and padding [1  1  1  1]

    3   'ReLU1'                   ReLU                  ReLU

    4   'Conv2'                   Convolution           64 3x3x64 convolutions with␣
↪stride [1  1] and padding [1  1  1  1]

    5   'BNorm2'                  Batch Normalization   Batch normalization with 64␣
↪channels

    6   'ReLU2'                   ReLU                  ReLU

    7   'Conv3'                   Convolution           64 3x3x64 convolutions with␣
↪stride [1  1] and padding [1  1  1  1]

    8   'BNorm3'                  Batch Normalization   Batch normalization with 64␣
↪channels

    9   'ReLU3'                   ReLU                  ReLU

   10   'Conv4'                   Convolution           64 3x3x64 convolutions with␣
↪stride [1  1] and padding [1  1  1  1]

   11   'BNorm4'                  Batch Normalization   Batch normalization with 64␣
↪channels

   12   'ReLU4'                   ReLU                  ReLU

   13   'Conv5'                   Convolution           64 3x3x64 convolutions with␣
↪stride [1  1] and padding [1  1  1  1]

   14   'BNorm5'                  Batch Normalization   Batch normalization with 64␣
↪channels

   15   'ReLU5'                   ReLU                  ReLU

   16   'Conv6'                   Convolution           64 3x3x64 convolutions with␣
↪stride [1  1] and padding [1  1  1  1]

   17   'BNorm6'                  Batch Normalization   Batch normalization with 64␣
↪channels

   18   'ReLU6'                   ReLU                  ReLU

   19   'Conv7'                   Convolution           64 3x3x64 convolutions with␣
↪stride [1  1] and padding [1  1  1  1]
   20   'BNorm7'                  Batch Normalization   Batch normalization with 64␣
↪channels
```

```
    21    'ReLU7'                  ReLU                  ReLU

    22    'Conv8'                  Convolution           64 3x3x64 convolutions with␣
↪stride [1  1] and padding [1  1  1  1]

    23    'BNorm8'                 Batch Normalization   Batch normalization with 64␣
↪channels

    24    'ReLU8'                  ReLU                  ReLU

    25    'Conv9'                  Convolution           64 3x3x64 convolutions with␣
↪stride [1  1] and padding [1  1  1  1]

    26    'BNorm9'                 Batch Normalization   Batch normalization with 64␣
↪channels

    27    'ReLU9'                  ReLU                  ReLU
    28    'Conv10'                 Convolution           64 3x3x64 convolutions with␣
↪stride [1  1] and padding [1  1  1  1]

    29    'BNorm10'                Batch Normalization   Batch normalization with 64␣
↪channels

    30    'ReLU10'                 ReLU                  ReLU

    31    'Conv11'                 Convolution           64 3x3x64 convolutions with␣
↪stride [1  1] and padding [1  1  1  1]

    32    'BNorm11'                Batch Normalization   Batch normalization with 64␣
↪channels

    33    'ReLU11'                 ReLU                  ReLU

    34    'Conv12'                 Convolution           64 3x3x64 convolutions with␣
↪stride [1  1] and padding [1  1  1  1]

    35    'BNorm12'                Batch Normalization   Batch normalization with 64␣
↪channels

    36    'ReLU12'                 ReLU                  ReLU

    37    'Conv13'                 Convolution           64 3x3x64 convolutions with␣
↪stride [1  1] and padding [1  1  1  1]

    38    'BNorm13'                Batch Normalization   Batch normalization with 64␣
↪channels

    39    'ReLU13'                 ReLU                  ReLU
    40    'Conv14'                 Convolution           64 3x3x64 convolutions with␣
↪stride [1  1] and padding [1  1  1  1]
```

```
   41   'BNorm14'                Batch Normalization   Batch normalization with 64␣
↪channels

   42   'ReLU14'                 ReLU                  ReLU

   43   'Conv15'                 Convolution           64 3x3x64 convolutions with␣
↪stride [1  1] and padding [1  1  1  1]
   44   'BNorm15'                Batch Normalization   Batch normalization with 64␣
↪channels

   45   'ReLU15'                 ReLU                  ReLU

   46   'Conv16'                 Convolution           64 3x3x64 convolutions with␣
↪stride [1  1] and padding [1  1  1  1]

   47   'BNorm16'                Batch Normalization   Batch normalization with 64␣
↪channels

   48   'ReLU16'                 ReLU                  ReLU
   49   'Conv17'                 Convolution           64 3x3x64 convolutions with␣
↪stride [1  1] and padding [1  1  1  1]

   50   'BNorm17'                Batch Normalization   Batch normalization with 64␣
↪channels

   51   'ReLU17'                 ReLU                  ReLU

   52   'Conv18'                 Convolution           64 3x3x64 convolutions with␣
↪stride [1  1] and padding [1  1  1  1]

   53   'BNorm18'                Batch Normalization   Batch normalization with 64␣
↪channels

   54   'ReLU18'                 ReLU                  ReLU

   55   'Conv19'                 Convolution           64 3x3x64 convolutions with␣
↪stride [1  1] and padding [1  1  1  1]

   56   'BNorm19'                Batch Normalization   Batch normalization with 64␣
↪channels

   57   'ReLU19'                 ReLU                  ReLU

   58   'Conv20'                 Convolution           1 3x3x64 convolutions with␣
↪stride [1  1] and padding [1  1  1  1]

   59   'FinalRegressionLayer'   Regression Output     mean-squared-error with␣
↪response 'Response'
```

We could **train** a new network for this task, but this is a rater complex topic, and a computationally intensive task. Therefore, we are not going to pursue it.

6.2 Deblurring images

The **blurring** of an image can be caused by many factors, the classical are

- a movement during the image capture process, e.g., a slight movement of the camera or even a micro-movement if long exposure times are used,
- an out-of-focus optics,
- the use of a wide-angle lens,
- an atmospheric turbulence for telescopic images,
- a too a short exposure time, e.g., the phenomena is very fast,
- the presence of scattered light and light distortion in confocal microscopy.

As many other things we have seen until now in this course, a **blurred image** can be described (at least approximately) by a *linear equation*:

$$\mathbf{y} = H\mathbf{x} + \mathbf{e},$$

where $\mathbf{x}$ is the image we would like to have, $\mathbf{y}$ is the image we got from the measurement procedure, $\mathbf{e}$ the noise vector, and H the *so-called* blur operator that encodes one of the **blurring** phenomena.

As we have done for the *noise* problem, we start by creating a fictitious *blurred* image. To reduce the computation time, we use a smaller example than the lion that is distributed with MATLAB

```
% First we read and visualize the image:
I = im2double(imread('peppers.png'));
figure(1)
subplot(1,2,1);
imshow(I);
title('Original Image');
% We create the operator H by using a PSF function
LEN = 31;
THETA = 11;
PSF = fspecial('motion',LEN,THETA);
Blurred = imfilter(I,PSF,'circular','conv');
figure(1);
subplot(1,2,2);
imshow(Blurred);
title('Blurred Image');
```



We need to discuss several aspects of these commands, first of all we need to describe how we build the H operator. This is done through the usage of a **P**oint **S**pread **F**unction (PSF), this function describes the response of the imaging system we are using when solicited by a point source. If we have in mind the model of a camera, this essentially tells us how a single ray of light (the point source) is distorted by the equipment we are using.

We produce a **synthetic response** by means of the `fspecial` function, its `help` produce

```
fspecial Create predefined 2-D filters.
   H = fspecial(TYPE) creates a two-dimensional filter H of the
   specified type. Possible values for TYPE are:

     'average'   averaging filter
     'disk'      circular averaging filter
     'gaussian'  Gaussian lowpass filter
     'laplacian' filter approximating the 2-D Laplacian operator
     'log'       Laplacian of Gaussian filter
     'motion'    motion filter
     'prewitt'   Prewitt horizontal edge-emphasizing filter
     'sobel'     Sobel horizontal edge-emphasizing filter
```

that proposes several type of filters. We selected a `motion blur`, that represents a linear motion of a camera by `LEN` pixels, with an angle of `THETA` degrees in a counter-clockwise direction. Then, we build the operator H by using the `imfilter` function. This operation needs the image on which we want to apply the the filtering, the PSF, the way in which we decide to model the boundaries of the image:

```
B = imfilter(A,H,OPTION1,OPTION2,...) performs multidimensional
filtering according to the specified options.  Option arguments can
have the following values:
```

(continues on next page)

(continued from previous page)

```
- Boundary options

    X             Input array values outside the bounds of the array
                  are implicitly assumed to have the value X.  When no
                  boundary option is specified, imfilter uses X = 0.

    'symmetric'   Input array values outside the bounds of the array
                  are computed by mirror-reflecting the array across
                  the array border.

    'replicate'   Input array values outside the bounds of the array
                  are assumed to equal the nearest array border
                  value.

    'circular'    Input array values outside the bounds of the array
                  are computed by implicitly assuming the input array
                  is periodic.
```

This is a **crucial** part, we need to guess how the image behaves out of the picture frame. This value are needed to close the model equation. For this test case we selected the `circular` option, that assumes that the values at the boundary are periodic. The last option decides how the PSF is transformed into the operator H

```
- Correlation and convolution

    'corr'        imfilter performs multidimensional filtering using
                  correlation, which is the same way that FILTER2
                  performs filtering.  When no correlation or
                  convolution option is specified, imfilter uses
                  correlation.

    'conv'        imfilter performs multidimensional filtering using
                  convolution.
```

Following the intuition behind the choice of the camera model, we decide for a `conv` process. At the end of this whole procedure, the variable `Blurred` contains the deteriorated image that we want to restore.

To **complete the problem** to a realistic solution, we need to also add some noise to the image. We have seen this procedure already in the previous section.

```
BlurredNoised = imnoise(Blurred,'gaussian',0,0.001);
figure(2)
imshow(BlurredNoised);
title('Image to be restored');
```



Now that we have the problem we want to solve, we can look at the routine proposed by MATLAB to solve this problem.

We use the `deconvwnr` function, this function requires as inputs the PSF, and the noise-to-signal-power ratio, let us start by looking at what happens if we miss the level of noise. As an example, let us suppose that we underestimate it, and decide that our image has no noise whatsoever:

```
nsr = 0;
Restored = deconvwnr(BlurredNoised,PSF,nsr);
imshow(Restored);
```



With this poor choice the algorithm is incapable of finding anything useful. We end up with a worse reconstruction than the image we had start with. In principle the problem we are trying to solve is nothing more than a linear system, the problem is that *we do not want to solve it completely*, if we do a complete solution we end up recovering only noise! Thus, many of the algorithms for this task require that the user provide an estimate of the noise. This is used to halt the solution procedure at a point in which the obtained solution makes sense. Thus, let us try with a better guess:

```
nsr = 0.001 / var(I(:));
Restored = deconvwnr(BlurredNoised,PSF,nsr);
figure(3)
imshow(Restored);
```



That is indeed a better result.

To have a complete introduction to this type of problems a good starting point is the book [Han10], there are then a number of books covering specific aspects ranging from the atmosphere [DTS10] to geophysical [Sun13, Zhd02].

CHAPTER

SEVEN

LINEAR ALGEBRA

We collect here some Linear Algebra concepts that appears here and there in these notes.

In all the following discussion the symbol $\mathbb{K}$ can be read as either the set of real numbers $\mathbb{R}$, or $\mathbb{C}$ the set of complex numbers.

Definition 8

An $m \times p$ **matrix** over $\mathbb{K}$ is an $m \times p$ rectangular array $A = (a_{i,j})_{\substack{i=1,\dots,m \\ j=1,\dots,p}}$ with entries in $\mathbb{K}$. The **size** of A is $m \times p$ and A is said to be square if $m = p$.

Two matrices A and B are said to be equal if they have the same size and $a_{i,j} = b_{i,j}\ \forall\, i, j$. We can build $\mathbb{K}^{m\times p}$ as the set of all the matrices of size $m \times p$ over the field $\mathbb{K}$ and define an addition and a scalar multiplication over $\mathbb{K}$ by $A + B \triangleq (a_{i,j} + b_{i,j})_{i,j}$ and the product with a scalar $c \in \mathbb{K}$ as $cA = (ca_{i,j})_{i,j}$, for all $c \in \mathbb{K}$.

Definition 9

Let $A = (a_{i,j}) \in \mathbb{K}^{m\times p}$ and let $\mathbf{b} \in \mathbb{K}^{p\times 1} \equiv \mathbb{K}^p$. The *matrix–vector product* of A and $\mathbf{b}$ is the vector $\mathbf{c} = A\mathbf{b}$ given elementwise by $c_i = \sum_{j=1}^{p} \mathbf{a}_{i,j} b_j$.

Let $A \in \mathbb{K}^{m\times p}$, $B \in \mathbb{K}^{p\times n}$ we define the *matrix product* of A and $B = [\mathbf{b}_{:,1}, \dots, \mathbf{b}_{:,n}]$ as $AB = [A\mathbf{b}_{:,1}, \dots, A\mathbf{b}_{:,n}]$ and, elementwise as

$$C = AB, \quad C = (c_{i,j}), \quad c_{i,j} = \sum_{k=1}^{p} a_{i,k} b_{k,j}.$$

The other operations that are interesting to look at for matrices and vectors are **norms**. In an *abstract* setting, norms are particular type of functions obeying the following characterization.

Definition 10

Let $(V, \mathbb{K})$ be a vector space with $\mathbb{K} = \mathbb{R}$ or $\mathbb{C}$ and $f : V \to \mathbb{R}$ a function such that

1. $f(\mathbf{v}) \geq 0\ \forall\, \mathbf{v} \in V$ and such that $f(\mathbf{v}) = 0$ if and only if $\mathbf{v} = \mathbf{0}$;
2. $f(\alpha\mathbf{v}) = |\alpha| f(\mathbf{v}), \forall\, \alpha \in \mathbb{K}, \forall\, \mathbf{v} \in V$;
3. $f(\mathbf{v} + \mathbf{w}) \leq f(\mathbf{v}) + f(\mathbf{w})\ \forall\, \mathbf{v}, \mathbf{w} \in V$.

The function f is a *norm* for the vector space V. The triple $(V, \mathbb{K}, f)$ is called a *normed space*.

Then examples of such norms for vectors are given by

Theorem 1

Let $V = \mathbb{C}^n$ or $\mathbb{R}^n$. If $p \geq 1$ we have that

$$\|\mathbf{x}\|_p \triangleq \left(\sum_{i=1}^{n} |x_i|^p \right)^{\frac{1}{p}}, \quad \forall\, \mathbf{x} \in V,$$

is a norm. Moreover,

$$\|\mathbf{x}\|_\infty \triangleq \max_i |x_i| = \lim_{p \to \infty} \|\mathbf{x}\|_p,$$

is a norm as well.

In general one can define also norm for matrices, e.g.,

Definition 11

Given a matrix $A \in \mathbb{K}^{m \times n}$, with $\mathbb{K} = \mathbb{R}$ or $\mathbb{C}$, the *Frobenius norm* of A is defined as,

$$\|A\|_F \triangleq \left(\sum_{i=1}^{m} \sum_{j=1}^{n} |a_{i,j}|^2 \right)^{\frac{1}{2}}.$$

The matrix norms associated to the p–norms from *Theorem 1* can be defined also as

$$\|A\|_p \triangleq \max_{\mathbf{v} \neq \mathbf{0}} \frac{\|A\mathbf{v}\|_p}{\|\mathbf{v}\|_p}.$$

Note that

$$\|A\|_1 \triangleq \max_{1 \leq j \leq n} \sum_{i=1}^{m} |a_{i,j}|, \quad \|A\|_\infty \triangleq \max_{1 \leq 1 \leq m} \sum_{j=1}^{n} |a_{i,j}|.$$

BIBLIOGRAPHY

[Vis00] Divakar Viswanath. Random Fibonacci sequences and the number $1.13198824…$. *Math. Comp.*, 69(231):1131–1155, 2000. URL: https://doi.org/10.1090/S0025-5718-99-01145-X, doi:10.1090/S0025-5718-99-01145-X.

[BW88] Douglas M. Bates and Donald G. Watts. *Nonlinear regression analysis and its applications*. Wiley Series in Probability and Mathematical Statistics: Applied Probability and Statistics. John Wiley & Sons, Inc., New York, 1988. ISBN 0-471-81643-4. URL: https://doi.org/10.1002/9780470316757, doi:10.1002/9780470316757.

[Lus03] David Lusseau. The emergent properties of a dolphin social network. *Proceedings of the Royal Society of London. Series B: Biological Sciences*, 270(suppl_2):S186–S188, 2003.

[RA15] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*. 2015. URL: https://networkrepository.com.

[vDKArguellesTico+14] Rene E van Dijk, Jennifer C Kaden, Araceli Argüelles-Ticó, Deborah A Dawson, Terry Burke, and Ben J Hatchwell. Cooperative investment in public goods is kin directed in communal nests of social birds. *Ecology letters*, 17(9):1141–1148, 2014.

[DTS10] Adrian Doicu, Thomas Trautmann, and Franz Schreier. *Numerical regularization for atmospheric inverse problems*. Springer Science & Business Media, 2010.

[Han10] Per Christian Hansen. *Discrete inverse problems*. Volume 7 of Fundamentals of Algorithms. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2010. ISBN 978-0-898716-96-2. Insight and algorithms. URL: https://doi.org/10.1137/1.9780898718836, doi:10.1137/1.9780898718836.

[Sun13] Ne-Zheng Sun. *Inverse problems in groundwater modeling*. Volume 6. Springer Science & Business Media, 2013.

[Zhd02] Michael S Zhdanov. *Geophysical inverse theory and regularization problems*. Volume 36. Elsevier, 2002.

PROOF INDEX

definition-0
definition-0 (*graphsandnetworks*), 53

definition-1
definition-1 (*graphsandnetworks*), 53

definition-2
definition-2 (*graphsandnetworks*), 53

definition-3
definition-3 (*graphsandnetworks*), 53

definition-4
definition-4 (*graphsandnetworks*), 53

definition-5
definition-5 (*graphsandnetworks*), 63

definition-7
definition-7 (*graphsandnetworks*), 65

frobenius_norm
frobenius_norm (*linearalgebra*), 88

matrix
matrix (*linearalgebra*), 87

norm
norm (*linearalgebra*), 87

products
products (*linearalgebra*), 87

thm:p-norms
thm:p-norms (*linearalgebra*), 87