

Lecture Notes on Numerical Methods for Engineering (Practicals)

Pedro Fortuny Ayuso

UNIVERSIDAD DE OVIEDO

E-mail address: fortunypedro@uniovi.es

© 2011–2016 Pedro Fortuny Ayuso

This work is licensed under the Creative Commons Attribution 3.0 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by/3.0/es/> or send a letter to Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.

Contents

Chapter 1. A primer on Matlab m-functions	5
Chapter 2. Ordinary Differential Equations	11
1. Cycling race profile	11
2. Numerical Integration of ODEs	16
Chapter 3. Ordinary Differential Equations (II)	23
1. The Lotka-Volterra model	23
2. Epidemic models (SIR)	27
3. Physical systems	28
Chapter 4. Numerical Integration	31
1. The simple quadrature formulas	31
2. Composite rules	33
Chapter 5. Interpolation	35
1. Linear interpolation	35
2. Cubic splines	36
3. Least Squares Interpolation	45
Chapter 6. Linear Systems of Equations	53
1. Exact methods	53
2. Iterative algorithms	56
Chapter 7. Approximate solutions to nonlinear equations	59
1. Bisection method	59
2. The Newton-Raphson method	60
3. The Secant method	62
4. The fixed-point method	63
Appendix A. Program structure	65

CHAPTER 1

A primer on Matlab m-functions

Along this course we are going to use constantly a Matlab/Octave tool for defining complex functions —more complex than the simple *anonymous functions*: m-files and functions defined in them.

An m-file is no more than a file whose name ends in `.m` and containing a number of Matlab/Octave commands. For example, the following, if called `trial1.m` might be called an m-file:

```
% This is just a simple file
e = exp(1);
b = linspace(-2,2,1000);
plot(b, e.\^b)
```

LISTING 1.1. A rather simple m-file

The power of m-files comes from two properties:

- If they are saved in the `PATH` directory (which includes the default `Documents/MATLAB`) then, running a command with the name of the file (without the `.m` extension) runs the contents of the file. That is, if after saving the above file, one runs

```
> trial1
```

then the plot of the function e^x for $x \in [-2, 2]$ using 1000 points is drawn.

- They can be used to define complex functions.

Instead of just a sequence of commands, one can define a function inside an m-file. Consider Listing 1.2, which defines a function returning the two maximum values of a list, in increasing order.

The structure of the file is the following:

- (1) Several comment lines (those starting with a `%` symbol). These comments describe what the function defined afterwards does.
- (2) A line like

```
function [y1, y2,...] = name(p1, p2,...)
```

where the word `function` appears at the beginning, then a list of names between square brackets (these are the *output variables*), an equal (=) sign, then *name of the function* (in

Listing 1.2, `max2`) and a list of parameters (the *input parameters*) between parentheses.

- (3) A sequence of lines of Matlab commands, which implement the function.
- (4) The `end` keyword at the end.
- (5) Finally, the name of the file must be the name of the function with a trailing `.m`. For Listing 1.2, it should be `max2.m`.

A file following all those rules defines a new Matlab/Octave command which can be used as any other command.

```
% max2(x)
% return the 2 maximum values in x, in increasing order
% if length(x) == 1, [-inf, x] is returned

function [y] = max2(x)

    % initialize: the first element is always greater than -inf
    y = [-inf, x(1)];

    % early exit test
    if(length(x) == 1)
        return;
    end

    % for each element, only do something if it is greater than y(1)
    for X=x(2:end)
        if(X > y(1))
            if(X > y(2))
                y(1) = y(2);
                y(2) = X;
            else
                y(1) = X;
            end
        end
    end
end
```

LISTING 1.2. A first m-file defining a function. Save it as `max2.m`

For example, if one saves the code in Listing 1.2 in a file named `max2.m` inside the `Documents/MATLAB` directory, then one can run the following commands:

```
> x = [-1 2 3 4 -6 9 -8 11]
x =

    -1     2     3     4    -6     9    -8    11
```

```
> max2(x)
ans =
```

```
9 11
```

which show that a new function, called `max2`, has been defined, and performs the instructions in the file `max2.m`.

In this course, we are going to use `m`-files and functions defined therein continuously, so the student is encouraged to write as many examples as possible. Learning to program in Matlab/Octave should be easy taking into account that the students have already undergone a course on Python.

REMARK 1. The main programming constructs we shall need are included in the Cheat Sheet (ask the professor for it if you do not have the link). They are the following:

- The `if...else` statement. It has the following syntax

```
if CONDITION
... % sequence of commands if CONDITION holds
else
... % sequence of commands if CONDITION does not hold
end
```

There are more possibilities (the `elseif` construct) but we are not going to detail them here.

- The `while` loop. Syntax:

```
while CONDITION
... % sequence of commands while CONDITION holds
end
```

will perform the commands inside the loop as long as the condition holds.

- The `for` loop. Syntax:

```
for var=LIST
... % sequence of commands
end
```

will assign sequentially to `var` each of the values of `LIST` and perform the commands inside the loop.

- Logical expressions. The conditions in `if` statements and `while` loops can be simple expressions (like `x < 3`, which means “x is less than 3”) or expressions built with logical operators: `and`, `or`, `not` or their *shortcut* versions `&&`, `||`, `~`.

Exercise 1: Implement a function `min3` which, given a list as input, returns its *three* minimal elements. Use `max2` defined above as a guide.

Exercise 2: Implement a function `increases` which, given a list as input, returns 1 if the list is in non-decreasing order and 0 if it is not. How would you implement this?

Exercise 3: Enhance the function of Exercise 2 so that it outputs two values: first of all, the same as `increases` and the length of the “increasing sequence” at the beginning of the input. Call it `increases2`. For example:

```
> [a,b] = increases2(7, -1, 2, 3 4)
a = 0
b = 1
> [u,v] = increases2(-1, 2, 5, 8, 9)
u = 1
v = 5
> [a,b] = increases2(3, 4, 5, -6, 8, 10)
a = 1
b = 3
```

Exercise 4: Define a function called `positive` which, given a list as input, returns two values: the number of *positive* (strictly greater than 0) elements in the list and the list of those elements. Examples:

```
> [a,b] = positive(-2, 3, 4, -5, 6, 7, 0)
a = 4
b = [3 4 6 7]
> [a,b] = positive(-1, -2, -3, -exp(1), -pi)
a = 0
b = []
> [a,b] = positive(4, 2, 1, 3 2)
a = 5
b = [4 2 1 3 2]
```

would you use a `while` loop or a `for`? Why?

Notice that functions returning several values can be requested to return any number of them. For example, the function `positive` defined in Exercise 4 might be requested to return just one value:

```
> positive(1, 2, -2, 0, 4)
3
> x = positive(-2, 1, 3, 7, 2)
x = 4
```

or it can be forced to return both. This requires using square brackets for the assignment of variables:


```
> [n x] = positive(pi, -2, 3, -5, 0)
n = 2
x = [pi 3]
```

If the function is called without assigning its output to a variable, it will return just one value. This will be relevant for many of the functions dealing with linear systems of equations (where one usually needs to know not just the final answer to a problem but the intermediate steps leading to it).

CHAPTER 2

Ordinary Differential Equations

This practical sessions are devoted to the numerical integration of ordinary differential equations. We shall mostly deal with one-variable problems but we might make an incursion into several variables, if time allows.

1. Cycling race profile

The most important idea to recall is that the derivative $y'(x)$ of a function of one variable, $y(x)$ at a point x_0 is *the slope of the graph of $y(x)$ at $(x_0, y(x_0))$* . This is so important that we shall take some time to chisel it on our minds.

Let $\mathbf{y}' = (y'_0, y'_1, \dots, y'_{n-1})$ denote the slopes of the road at different points in a bicycle race and $\mathbf{x} = (x_0, x_1, \dots, x_n)$ the horizontal coordinates of each point. Notice that \mathbf{x} is not the list of km marks at each point but the OX -axis coordinate of each point. Notice also that \mathbf{y}' has one coordinate less than \mathbf{x} .

If the race starts at height y_0 , how would one compute the approximate heights at each point x_i for $i = 1, \dots, n$?

This is an easy example with several possible solutions (depending on the reader's preference). The most obvious way to tackle this problem is by assuming that the slope on each interval $[x_i, x_{i+1})$ is constant and has value y'_i . This way, the height at x_1 would be computed as $y_1 = y_0 + y'_0(x_1 - x_0)$, because the line passing through (x_0, y_0) with slope y'_0 has equation

$$y = y_0 + y'_0(x - x_0).$$

From here one can now compute the approximate height at x_2 , using the same reasoning: $y_2 = y_1 + (x_2 - x_1)y'_1$, and iteratively,

$$y_i = y_{i-1} + (x_i - x_{i-1})y'_{i-1}, \quad \text{for } i = 1, \dots, n.$$

This shows clearly why the initial height y_0 is necessary: without it there is no way to compute y_1 and hence, y_i for $i \geq 1$.

Example 1 A simple example with just 5 nodes. Let a road have the following list of slopes: (+4%, -2%, +7%, +12%) at the horizontal

coordinates (in km): $(0, 2, 5, 7)$. The road reaches up to km 8 on the OX axis. The race starts at a height of $850m$. Draw a profile of the race.

This example can be done by hand:

- (1) The first stretch is $2km$ long horizontally and has a slope of $+4\%$. This means that, if the slope is constant, the road goes up by $2km \times 4\% = 80m$. As the race starts at $850m$, after this stretch, the road will be approximately $850m + 80m = 930m$ high.
- (2) The second stretch is $3km$ long horizontally and its slope is -2% , so that the road goes *down* by $3km \times 2\% = 60m$. Hence, after this stage the road is $930m - 60m = 870m$ high.
- (3) The third stage is $2km$ long and has slope $+7\%$. This gives $140m$ up, so the road ends at $870m + 140m = 1010m$.
- (4) Finally, $1km \times 12\% = 120m$, so that the road ends at $1010m + 120m = 1130m$.

The approximate profile is plotted in Figure 1. —

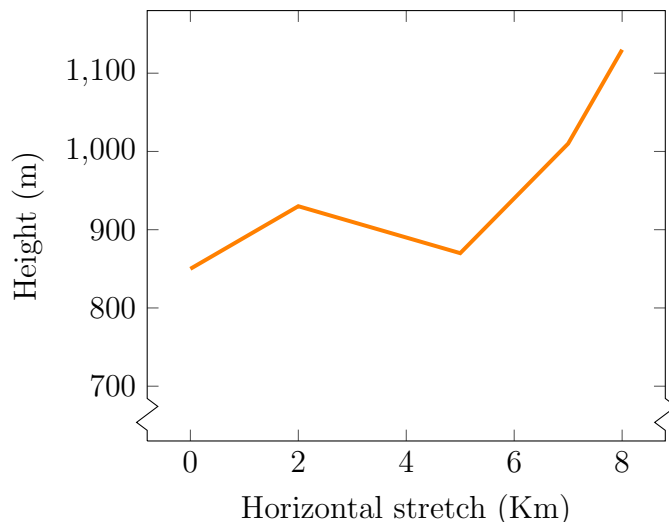


FIGURE 1. Approximate profile of a short race.

Example 2 The same example, with many more data. Construct \mathbf{x} as a vector with 25 components between 0 and 200. Construct now a random vector \mathbf{y}' with values between -15% and 15% (which are normal slopes for a road). Choose a height y_0 as the starting value. Compute the successive heights for each x_i and draw the profile of the stage.

A solution to this can be found in Listing 2.1

```
% A 'stage' of a cycling race, using random slopes.

% Problem setup.
x = linspace(0, 200, 25);

% Explanation of the next line:
% (Notice that there is ONE slope less than x-coordinates!)
% Take a random value (well, 24 of them) between 0 and 1
% Scale them so that they are between 0 and .30
% Subtract .15 so that they are between -.15 and +.15
yp = rand(1,24) * .30 - .15;

% Initial height (choose your own)
y0 = 870;

% Approximate stage profile.
% 1) List of "rises" or "descents"
h = diff(x).*yp;

% 2) Create the vector of heights, "empty" but for the first one:
y = zeros(1, 25);
y(1) = y0;

% 3) For each "step" do:
%     Add to the list of heights the last one computed
%     plus the corresponding difference
for s = 1:length(h)
    y(s+1) = y(s) + h(s);
end

% 4) Finally, plot the profile of the stage
plot(x,y);
```

LISTING 2.1. Approximate profile of a cycling race, first version.

A possible profile is plotted in Figure 2. —

Of course, the method explained is one way to solve the stage profile problem approximately. There are more (although with the data that is available, there are not *many* more which are reasonable).

Example 3 The same problem can be tackled differently. Notice that in Example 2 we are using just the slope *at the beginning of the interval*, which may be too little information. One might think “why use the left endpoint and not the right one?”. As a matter of fact, a more reasonable solution would be to somehow use the information at both endpoints for each interval. Instead of taking y'_{i-1} or y'_i as the

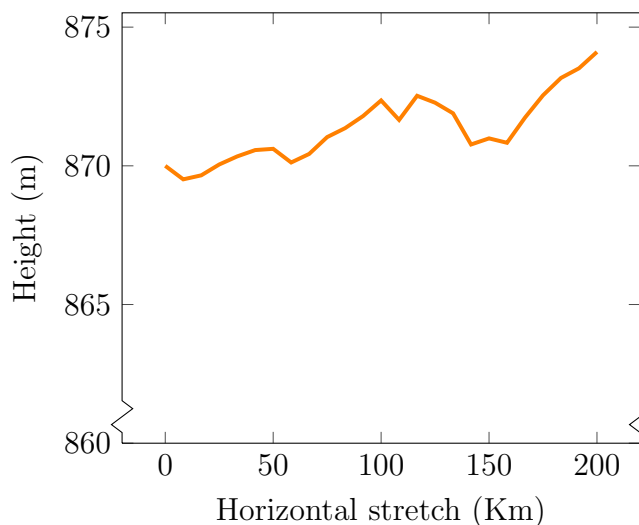


FIGURE 2. Approximate profile of a long race.

slope for interval $[x_{i-1}, x_i]$, one could use the mean value as the “mean slope” on that interval:

$$\tilde{y}'_{i-1} = \frac{y'_{i-1} + y'_i}{2}$$

and compute the height at step i as

$$y_i = y_{i-1} + (x_i - x_{i-1})\tilde{y}'_{i-1}.$$

Notice (and this is important) that in order to perform these calculations, we *need to know* the slope at the last point, so that \mathbf{y}' must have as many components as \mathbf{x} for this method to work.

This leads to the code of Listing 2.2. —

```
% A 'stage' of a cycling race, using random slopes.

% Problem setup.
x = linspace(0, 200, 25);

% Explanation of the next line:
%   Take a random value (well, 25 of them) between 0 and 1
%   Scale them so that they are between 0 and .30
%   Subtract .15 so that they are between -.15 and +.15
yp = rand(1,25) * .30 - .15;

% Initial height (choose your own)
y0 = 870;
```

```

% Approximate stage profile.
% 1) List of "rises" or "descents".
%   We have to use the mean value between y(i-1) and y(i)...
%   See how 'end-1' serves as an index!
yp_means = (yp(1:end-1) + yp(2:end))/2;
%   h is the vector of 'vertical differences'
h = diff(x).*yp_means;

% 2) Create the vector of heights, "empty" but for the first one:
y = zeros(1, 25);
y(1) = y0;

% 3) For each "step" do:
%       Add to the list of heights the last one computed
%       plus the corresponding difference
for s = 1:length(h)
    y(s+1) = y(s) + h(s);
end
% This loop is not 'right', there is a simpler way to do the same:
% y(2:end) = y(1:end-1) + h;
% Which is more 'Matlab'-ish and, in fact, clearer.

% 4) Finally, plot the profile of the stage
plot(x,y);

```

LISTING 2.2. Approximate profile of a cycling race,
“mean value” version.

Exercise 5: Using the codes of Listings 2.1 and 2.2, compare the solutions to the same problem using both methods; “compare” both analytically (using absolute and relative differences) and graphically.

Exercise 6: Write two m-files, one for each of the methods in examples 2 and 3. In each file, a function should be defined, taking as arguments two vectors of the same length, x and yp and a real number y_0 . The output should be a vector y containing the heights of the profile at each point in x . Call them `profile_euler.m` and `profile_mean.m`.

Use them several times with the same data and compare the plots. Which is smoother? Why?

A sample `profile_euler.m` file might read as Listing 2.3

```

% profile_euler(x, yp, y0):
%
% Given lists of x-coordinates and yp-slopes and an initial height y0,
% return the heights at each stretch (coordinate x) of a road having
% slopes yp, starting at height y0.
function [y] = profile_euler(x, yp, y0)

```

```

y = zeros(size(x));
y(1) = y0;
for s = 1:length(x)-1
    y(s+1) = y(s) + yp(s)*(x(s+1) - x(s));
end
end

```

LISTING 2.3. Sample code for `profile_euler.m`.

2. Numerical Integration of ODEs

The previous section was just an introduction to the problem of numerical integration of Ordinary Differential Equations (ODE from now on). As we saw in class, we shall consider only¹ ODEs of the form

$$y' = f(x, y).$$

We are now going to translate this expression into ordinary language.

First of all, the derivative of a function $y(x)$ *the slope of the graph* $(x, y(x))$ *at each point*: let $y(x)$ be a differentiable function and x_0 a real number. Then $y'(x_0)$ is exactly the *slope* (as the slope of a road, the same concept) of the graph of $y(x)$ at $(x_0, y(x_0))$.

In Figure 3 the graph of the function $y(x) = \sin(x)$ has been plotted together with the tangent line at $(\frac{2\pi}{6}, \frac{\sqrt{3}}{2})$. Notice how the slope of this line is 0.5 (as a matter of fact, notice how it goes up by 2 units vertically along a 4 units long interval) which is exactly the value of $y'(x) = \cos(x)$ at $x_0 = \frac{2\pi}{6}$. This should reinforce the idea that “derivative means slope.”

With this mindset, the expression

$$y' = f(x, y)$$

can only have an interpretation: “the function $y(x)$ is such that its slope at each point $(x_0, y(x_0))$ is exactly $f(x_0, y(x_0))$.” Briefly stated, “the slope of the curve $y(x)$ at x is $f(x, y)$.”

As in the case of the cycling race, giving the slopes of the road at each point is not enough to compute the heights: one needs an initial height in order for the problem to have a definite solution. The same happens with an ODE: the single statement $y' = f(x, y)$ cannot be enough to provide a solution. One needs a piece of data more: the “initial height” of the graph of $y(x)$. This gives rise to the notion of *initial value problem*.

¹Or rather, *mainly*.

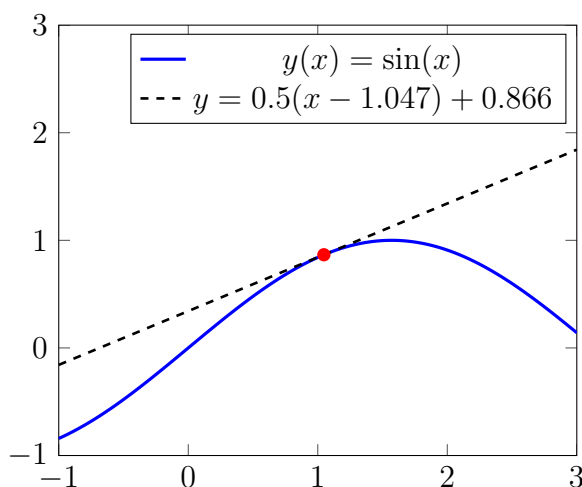


FIGURE 3. Graph of $y(x) = \sin(x)$ and its tangent at $(\frac{2\pi}{6}, \frac{\sqrt{3}}{2})$.

DEFINITION 1. An initial value problem is an ODE $y' = f(x, y)$ together with a pair (x_0, y_0) called the *initial condition* or *initial value*.

Once an initial value is given for $y(x_0)$ at some x_0 , the ODE $y' = f(x, y)$ has a unique solution².

Example 4 The initial value problem

$$y' = y, \quad y(0) = 1,$$

has as solution the function $y(x) = e^x$. Check this.

If instead of $y(0) = 1$ one has $y(0) = K$, then the solution to the corresponding initial value problem is

$$y(x) = Ke^x.$$

What happens if the initial value is $y(1) = 0.5$. Is it necessary that $x_0 = 0$ or can one compute the solution anyway? —

As the reader will have already noticed, in order to state an initial value problem, one needs the following data:

- (1) The function $f(x, y)$, of two real variables.
- (2) The pair (x_0, y_0) , that is, two real values, one for the x and another one for the corresponding y .

However, for a numerical approach, one also needs the *network of x -coordinates* on which approximations of $y(x)$ will be computed. As in the cycling examples above, this may be a vector \mathbf{x} of “horizontal

²Under natural conditions which are outside the scope of the practicals.

positions.” Thus, in order to compute a numerical approximation to the solution $y(x)$, one requires

(3) A vector \mathbf{x} of x -coordinates on which to approximate $y(x)$.

The “solution” to the numerical integration of the ODE will be the list of values $y(x)$ for each $x \in \mathbf{x}$. The first one will be, obviously, y_0 , the remaining ones will be approximations to the true solution.

2.1. Euler’s method. The first naive numerical method is Euler’s algorithm which follows literally Example 2. Let n be the length of the vector \mathbf{x} . Then one can describe Euler’s method as Algorithm 1, which is as easy as it gets. Notice that x_0 and y_0 are *part of the data*.

Algorithm 1 Euler’s method.

```

for  $i = 1 \dots n$  do
     $y_i = y_{i-1} + (x_i - x_{i-1})f(x_{i-1}, y_{i-1})$ 
end for
return  $(y_0, y_1, \dots, y_n)$ 

```

The statement inside the **for** loop is exactly the same as for the cycling race profile: the height at position x_i is approximated as the height at x_{i-1} plus the slope at this point (which is $f(x_{i-1}, y_{i-1})$) times the horizontal step (which is $x_i - x_{i-1}$). Finally, the algorithm returns the vector of heights.

Implementing the above as a Matlab **m**-function should be straightforward. However, we are including the complete code in Example 5 for the benefit of the reader. We have called the function **euler** and, as a consequence, the file must be called **euler.m**.

Example 5 Given a function f —which we shall assume is given as an *anonymous function*—, a vector \mathbf{x} of horizontal positions and an initial value y_0 , an **m**-file which implements Euler’s method for numerical integration of ODEs as a Matlab function may contain the code in Listing 2.4. We emphasize that *the file name name must be euler.m*, otherwise it would not work. —

```

% Euler’s method for numerical integration of ODEs
% INPUT:
% 1) an anonymous function f
% 2) a vector of x-positions (including x0 as the first one)
% 3) an initial value y0, corresponding to x0

% OUTPUT:
% a vector y of values of the approximate solution at 'x'

```

```

function [y] = euler(f, x, y0)
    % first of all, create 'y' with the same length as x
    y = zeros(size(x));
    % and store the initial condition in the first position
    y(1) = y0;

    % Run Euler's loop
    for s = 2:length(x)
        y(s) = y(s-1) + f(x(s-1), y(s-1)).*(x(s) - x(s-1));
    end
end

```

LISTING 2.4. Matlab code for Euler's numerical integration method.

Exercise 7: Use the function `euler` just implemented to solve numerically the following initial value problems. Use as many points as you wish (not more than 100, though) when not specified. Plot the solutions as you compute them.

- For $x \in [0, 1]$, solve $y' = 2x$ with $y(0) = 1$. Use 10 points. Plot, on the same graph, the true solution $y(x) = x^2 + 1$.
- For $x \in [0, 1]$, solve $y' = y$ with $y(0) = 1$. Use 15 points. Plot, on the same graph, the true solution $y(x) = e^x$.
- For $x \in [0, 1]$, solve $y' = xy$ with $y(0) = 1$. Plot, on the same graph, the true solution $y(x) = e^{\frac{x^2}{2}}$.
- For $x \in [-1, 1]$, solve $y' = x + y$ with $y(-1) = 0$.
- For $x \in [-\pi, 0]$, solve $y' = \cos(y)$ with $y(-\pi) = -1$.
- For $x \in [1, 3]$, solve $y' = y - x$ with $y(1) = -1$.

However, Euler's method is not exactly *the best way* to approximate solutions to ODEs. As the first three initial value problems in Exercise 7 show, solutions computed using Euler's method are usually below the true solution if this is convex (or above it, when concave). This is because convexity means $y'(x)$ is an increasing function (and concavity means $y'(x)$ is decreasing), so that Euler's algorithm is always short of the true solution. This lack of precision can be overcome partly using an intermediate point instead of the left endpoint of the interval, which is *modified Euler's method*.

2.2. Modified Euler's method. Instead of using the value of $f(x, y)$ at the left endpoint of each interval, one can perform the following "improvement:"

- (1) Assume y_{i-1} has been computed.

- (2) Let $k = f(x_{i-1}, y_{i-1})$ (the slope of Euler's method).
- (3) Let $\tilde{x} = \frac{x_i + x_{i-1}}{2}$ be the midpoint of $[x_{i-1}, x_i]$.
- (4) Let $z = \frac{k}{2}(x_i - x_{i-1})$ be half the vertical step corresponding to Euler's approximation.
- (5) Let $r = f(\tilde{x}, y_{i-1} + z)$ be the slope described by $f(x, y)$ at the midpoint of Euler's approximation, that is: (\tilde{x}, z) .
- (6) Finally, $y_i = y_{i-1} + r(x_i - x_{i-1})$ is the next approximate value of the solution.

Although the description seems confusing, the above method can be described as follows: "Use Euler's method to compute the next *midpoint*, then use the value of f at this midpoint as the slope at the present point." So, instead of using the slope at the left endpoint, one uses the slope at some "midpoint" in the hope that it will give a better approximation. As a matter of fact, this happens in most cases. Formally, the above could be described with Algorithm 2.

Algorithm 2 Modified Euler's method.

```

for  $i = 1 \dots n$  do
   $k = f(x_{i-1}, y_{i-1})$ 
   $\tilde{x} = \frac{x_{i-1} + x_i}{2}$ 
   $z = \frac{k}{2}(x_i - x_{i-1})$ 
   $r = f(\tilde{x}, y_{i-1} + z)$ 
   $y_i = y_{i-1} + r(x_i - x_{i-1})$ 
end for
return  $(y_0, \dots, y_n)$ 

```

Notice how $f(x, y)$ needs to be evaluated twice in this new algorithm. This is essentially what gives the enhanced accuracy of this method. As a general rule, the more evaluations of $f(x, y)$ are carried out (reasonably), the more accurate a method will be, and vice versa: the more accurate a method, the more evaluations of $f(x, y)$ it will require. Efficient algorithms for numerical integration of ODEs balance speed and accuracy.

Exercise 8: Implement modified Euler's method in an `m`-file, call it `modified_euler.m` and use it to solve the initial value problems in Exercise 7. Compare (graphically and analytically) both approximate solutions and the exact one in the cases where this is given. Is this method better or worse? Is it always so or just some times? —

2.3. Heun’s method (“improved Euler”). The third method for numerical integration of ODEs which we shall explain is the equivalent of the *mean-value* algorithm for the cycling race explained in Example 3: use the mean of the slopes at the left and right endpoints of each interval. This is known as Heun’s or *improved Euler’s* method.

However, there is a difference with respect to the cycling race example. In the race, *we already know the slope at the right endpoint*: as a matter of fact, we know the slopes at each point on the x -axis. On the contrary, if we are given the differential equation

$$y' = f(x, y)$$

and the initial value (x_0, y_0) , we can compute the slope at this point $f(x_0, y_0)$ but the question arises: *what is the “next point”?* There is no such thing because the solution to the ODE is unknown. We only know the next x -coordinate, x_1 but we lack the y -coordinate (otherwise we would know the solution to the problem). We need to *guess* (or, more precisely, to *predict*) a value for y_1 and then “correct” it somehow. Heun’s method follows the following mental process:

- (1) Start at (x_0, y_0) .
- (2) Compute (x_1, \tilde{y}_1) using Euler’s method as a “guess.” This requires using $k_1 = f(x_0, y_0)$ as slope.
- (3) Compute $k_2 = f(x_1, \tilde{y}_1)$, the slope at Euler’s guess.
- (4) Instead of using either k_1 or k_2 , take the mean value of both slopes, $k = (k_1 + k_2)/2$.
- (5) Use k as the slope from (x_0, y_0) . That is,

$$y_1 = y_0 + k(x_1 - x_0).$$

An example should illustrate things a bit.

Example 6 Consider the initial value problem (IVP)

$$y' = x - y, \quad y(2) = 0.$$

and let $\mathbf{x} = (2, 2.25, 2.5)$ be a sequence of x -coordinates. Let us use Heun’s method to find an approximate solution to the IVP at $x = 2.25$ and $x = 2.5$.

We need to perform two steps of Heun’s method. We shall detail the first one and run through the second one.

First step: $x_0 = 2, y_0 = 0$.

- Using Euler’s method, one has $k_1 = f(x_0, y_0) = 2$ (this point (x_0, y_0) is the initial condition) and so, $\tilde{y}_1 = 0 + 2 \times 0.25 = 0.5$.
- Using the previous data, $k_2 = f(2.25, 0.5) = 1.75$.
- The mean value of k_1 and k_2 is $k = 1.875$.
- Finally, $y_1 = y_0 + k(x_1 - x_0) = 0 + 1.875 \times 0.25 = 0.46875$.

Hence, $(x_1, y_1) = (2.25, 0.46875)$.

Second step: $x_1 = 2.25, y_1 = 0.46875$.

From this data, we get $\mathbf{k}_1 = f(x_1, y_1) = 1.7812$ and $\tilde{\mathbf{y}}_2 = 0.46875 + 1.7812 \times 0.25 = 0.91405$, so that $\mathbf{k}_2 = f(x_2, \tilde{\mathbf{y}}_2) = 1.5859$. This gives $\mathbf{k} = 1.6835$ and the result is $(x_2, y_2) = (2.5, 0.88962)$.

Taking into account that the exact solution to the problem is $y(x) = x - e^{2-x} - 1$, which gives $y(2.5) = 0.89347$, the relative error incurred is $\frac{|0.89347 - 0.88962|}{0.89347} \simeq 0.004$, which is rather small (even more if we consider that the steps are quite large, 0.25 units each). —

Algorithm 3 is a more formal expression of the method.

Algorithm 3 Heun's method, also called "improved Euler's" method.

```

for  $i = 1 \dots n$  do
     $k_1 = f(x_{i-1}, y_{i-1})$ 
     $\tilde{y}_i = y_{i-1} + k_1(x_i - x_{i-1})$ 
     $k_2 = f(x_i, \tilde{y}_i)$ 
     $k = \frac{k_1 + k_2}{2}$ 
     $y_i = y_{i-1} + k(x_i - x_{i-1})$ 
end for
return  $(y_0, \dots, y_n)$ 

```

Exercise 9: In an m-file, implement Heun's method with a function called `heun`. (Remember that the file must then be named `heun.m`). Use it to find approximate solutions to the IVPs of Exercise 7 and compare these to the ones found using Euler's and Modified Euler's. Which are better? Always? When? —

Exercise 10: Use a spreadsheet (like Excel or LibreOffice Calc) to implement Euler's method. Explain how it might be done (and do it) for the IVPs of Exercise 7. Compare the results with the ones produced by Matlab/Octave. Can Modified Euler's and Heun's methods be implemented in a spreadsheet? If they can, do so for at least one of them.

Use the charting utility of the spreadsheet to plot the graphs of the solutions. —

CHAPTER 3

Ordinary Differential Equations (II)

We are going to work out some examples in this chapter.

1. The Lotka-Volterra model

The first non-trivial differential equation describing a biological system which we are going to study is called the “Lotka-Volterra” equation and is used to model an environment in which two species live, one which behaves as a prey and the other as its predator.

Let $x(t)$ denote the population of a “prey” species at time t , and $y(t)$ the population of the “predator” species. The differential equation describing this model is based on the following (quite simplistic) assumptions:

- (1) The prey species breeds proportionally to its number.
- (2) A prey dies only as a consequence of being eaten by some predator, with some constant probability.
- (3) The predator species dies proportionally to its number.
- (4) Predators only breed in proportion to their eating the preys.

From item 1 we infer that there is a number $\alpha > 0$ such that

$$\dot{x}(t) = \alpha x(t) + \dots$$

for some expression instead of the dots. This is an exponential increase in the number of preys (apart from the interaction with the predators).

From item 3 we conclude that there is a number $\gamma > 0$ such that

$$\dot{y}(t) = -\gamma y(t) + \dots$$

for some other (different from the above) expression instead of the dots. This gives an exponential decrease in the number of predators (in the absence of feeding).

It is easy to show that the probability of a predator meeting a prey at some point in space is proportional to the product of the number of predators and preys. Because not every meeting ends up in a prey being eaten, we model the probability of this event as $\beta x(t)y(t)$ (with $\beta > 0$) and because not every feeding of a predator gives rise to breeding, we model the probability of breeding (for predators) as $\delta x(t)y(t)$ for some

$\delta > 0$. Hence, we can substitute the dots above by the corresponding values and get the Lotka-Volterra differential equation:

$$(1) \quad \begin{aligned} \dot{x}(t) &= \alpha x(t) - \beta x(t)y(t) \\ \dot{y}(t) &= -\gamma y(t) + \delta x(t)y(t) \end{aligned}$$

Example 7 Model a Lotka-Volterra system with parameters $\alpha = 0.8$, $\beta = 0.4$, $\gamma = 2$, $\delta = 0.2$ and initial populations $x(0) = 18$, $y(0) = 3$. Use Euler's method to compute an approximation with a timestep of 0.1 units and compute up to 12 seconds.

We first do it by hand and then shall try to write a general program. Listing 3.1 is a complicated way to work out this example and plot the evolution of both populations in time.

```
% Lotka-Volterra simulation, first version
% Time from 0 to 12 seconds
t=[0:.1:12];
% Create an empty list of the adequate size for both variables
x=zeros(size(t));
y=zeros(size(t));
% Initial values
x(1)=18;
y(1)=3;

% The differential equation (Notice 3 variables)
xp = @(t,x,y) 0.8*x - 0.4*x*y;
yp = @(t,x,y) -2*y + 0.2*x*y;

% Do the Euler step for each time
for k=1:length(t)-1
    x(k+1) = x(k) + xp(t(k),x(k),y(k)) * (t(k+1) - t(k));
    y(k+1) = y(k) + yp(t(k),x(k),y(k)) * (t(k+1) - t(k));
end

% Finally, plot both species on the same graph
plot(t,x)
hold on
plot(t,y,'r')
```

LISTING 3.1. A first approximation to the Lotka-Volterra equations.

Notice how the populations have an increasing and decreasing behaviour, with a shift.

Also, with some effort one can verify approximately that the critical points of $x(t)$ are reached when $y(t) = \gamma/\delta$ and that the critical points of $y(t)$ happen when $x(t) = \alpha/\beta$. (How would you do this? It is not so easy but not so difficult either).

However, it is known that the Lotka-Volterra system is periodic (this is something known, not something that we have proved in the theory classes) and the approximation we have plotted is not periodic (if one goes on plotting, the graphs become more and more separated and spiky). This anomaly is due also to the intrinsic imprecision of Euler's method. —

It should be easy to realize that the process above can be simplified if one writes a function to perform Euler's algorithm with ordinary differential equations in several variables.

First of all, the expression

$$\begin{aligned}\dot{x}(t) &= \alpha x(t) - \beta x(t)y(t) \\ \dot{y}(t) &= -\gamma y(t) + \delta x(t)y(t)\end{aligned}$$

can be written in vector form as

$$\begin{pmatrix} \dot{x}(t) \\ \dot{y}(t) \end{pmatrix} = \begin{pmatrix} f_1(t, x, y) \\ f_2(t, x, y) \end{pmatrix}$$

for adequate values of f_1 and f_2 . In general, it does not need to be a 2-dimensional vector: it can be of any size. And, to make things even more general, one might write the above as:

$$\begin{pmatrix} \dot{x}_1(t) \\ \vdots \\ \dot{x}_n(t) \end{pmatrix} = F(t, x_1, \dots, x_n)$$

where F is a *column vector function* of n components. Thus, in order to describe the initial value problem, one needs:

- An n -component *column vector function* of $n + 1$ variables.
- The n initial values, one for each x_i .

Notice that we work with column vectors, which is the way one usually writes ordinary differential equations on paper.

One might define then a function `eulervector` which receives:

- an anonymous function returning a column vector (the F above),
- a vector of time-positions and
- a column vector of initial positions

and which returns a list of column vectors with the same length as the list of time-positions (that is, a matrix of n rows, as many as variables and l columns, as many as time-positions).

Example 8 The code in Listing 3.2 shows a possible implementation of the function `eulervector` described above.

```

% Euler's method for numerical integration of ODEs, vector version
% INPUT:
% 1) an anonymous function f(t, x)
%    of two variables:
%    t: numerical
%    x: column vector
% 2) a vector T of t-positions (including t0 as the first one)
% 3) a column vector x0 of initial values of x for t=t0

% OUTPUT:
% a matrix of as many rows as x0 and as many columns as T above,
% which approximate the solution to the (vector) ODE given by f.
function [y] = eulervector(f, x, y0)
    % first of all, create 'y' with the adequate size
    y = zeros(length(y0),length(x));
    % and store the initial condition in the first position
    y(:,1) = y0;

    % Run Euler's loop
    for s = 2:length(x)
        y(:,s) = y(:,s-1) + (x(s) - x(s-1)) * f(x(s-1),y(:,s-1));
    end
end
end

```

LISTING 3.2. A vector implementation of the Euler method.

Notice in the code that the only difference with our previous implementation of Euler's algorithm, as in Listing 2.4, is the explicit apparition of the rows in the y vector, both at the beginning, in the line

```
y = zeros(length(y0), length(x));
```

and all the other times, with the colon in the expressions $y(:, \dots)$.

Using this function, the Lotka-Volterra model of Example 7 can be worked out with the code in Listing 3.3.

```

% Define the function (same parameters)
% Notice that because x is a vector (albeit a column one),
% one can reference each component with a single index:
f = @(t, x) [0.8*x(1) - 0.4*x(1)*x(2) ; -2*x(2) + 0.2*x(1)*x(2)];

% Set up the time
T = [0:.1:12];
% Initial conditions
X0 = [18; 3];
% Solve
X = eulervector(f, T, X0);
% Plot. Notice how Matlab plots each row using different colours

```

```
plot(T, X)
```

LISTING 3.3. Lotka-Volterra with `eulervector`.

It should be noted that one no longer uses two different variables, \mathbf{x} and \mathbf{y} , but a single “column vector” \mathbf{x} , and references each component using the appropriate index. So, when defining \mathbf{f} , instead of \mathbf{x} one uses $\mathbf{x}(1)$ and instead of \mathbf{y} , one uses $\mathbf{x}(2)$. Also, the solution is stored in a single variable (\mathbf{X} in the example) which (in the example) has two rows, one for each component of the system.

Finally, notice how Matlab plots each row in a matrix with a different colour, as though it were a list. This makes unnecessary the use of `hold on`. —

Exercise 11: Write a function `heunvector` which implements Heun’s method for vector differential equations. It should be obvious that the only difference with Listing 3.2 should be changing the following line

$$\mathbf{y}(:,\mathbf{s}) = \mathbf{y}(:,\mathbf{s}-1) + (\mathbf{x}(\mathbf{s}) - \mathbf{x}(\mathbf{s}-1)) * \mathbf{f}(\mathbf{x}(\mathbf{s}-1),\mathbf{y}(:,\mathbf{s}-1));$$

for something a bit more complicated (the Heun step of “going forward like Euler, computing the velocity and then using the mean value of both velocities at the starting point.”) —

Exercise 12: Using the function `heunvector` which has just been defined, plot the approximate solution to the Lotka-Volterra equation of Example 7. Plot, also, the solution given by Euler’s method and compare them. Which looks more reasonable? Why? What are the absolute and relative differences after 12 seconds?

Verify (somehow) that the maxima and minima of each variable correspond to the quotients of the parameters of the other one. How could you do this?

You should notice how Heun’s solution looks more periodic than Euler’s. Actually, this reflects the true solution to the system more accurately, as it is known that the solutions to the Lotka-Volterra equations are periodic, indeed. —

2. Epidemic models (SIR)

Exercise 13: The *SIR* epidemic model for an infectious disease follows the differential equation:

$$\begin{aligned}\dot{S}(t) &= -\alpha S(t)I(t) \\ \dot{I}(t) &= -\beta I(t) + \alpha S(t)I(t) \\ \dot{R}(t) &= \beta I(t)\end{aligned}$$

for some positive numbers α and β . The variables S, I and R stand for “susceptible”, “infected” and “removed.” Use the Matlab function `heunvector` defined in Exercise 11 to plot several models of this equation, for different initial values and parameters. An interesting example is $S(0) = 1, I(0) = 0.001, R(0) = 0$ and $\alpha = 0.1, \beta = 0.05$, time from 0 to 1500 using 2000 points, for example. Compare the evolution of that system with another one having $\alpha = 0.05$ and $\beta = 0.01$. Does the system behave differently if $\alpha > \beta$ or if $\alpha < \beta$? In what way?

Compare also the difference between using `heunvector` and using `eulervector`. Which seems more precise? Why? —

Exercise 14: Modify the model in Exercise 13 to allow that some of the infected people become susceptible, with a coefficient γ . Compare this model to the previous one, using `heunvector`. —

Exercise 15: Allow for births in the model of Exercise 14 (i.e. $S(t)$ increases proportionally to the sum of $S(t), I(t)$ and $R(t)$) with some small coefficient δ). Compare this model with the ones of Exercises 14 and 13. Explain the plots and the (remarkable) difference between $R(t)$ in both previous exercises and this one. —

Exercise 16: Allow for deaths in the model of Exercise 15: let each of $S(t), R(t)$ and $I(t)$ decrease with different probabilities ϵ_1, ϵ_2 and ϵ_3 , with $\epsilon_1 = \epsilon_3$ and $\epsilon_2 > \epsilon_1$. Does the model change a lot? Why? Compare with the previous ones. —

3. Physical systems

Example 9 A simple pendulum without friction can be simulated using the following (polar) equation

$$l\ddot{\theta} = -g \sin(\theta)$$

where θ is the angle with respect to the vertical and l is the (constant) length of the pendulum. Notice how the mass of the weight is irrelevant. We can use `heunvector` to compute an approximation to the solution of this equation for example, using $l = 1, g = 9.81$ and two different initial conditions: $\theta(0) = \pi/4$ and $\theta(0) = \pi/3$ with initial velocities 0 in both cases, with time going from 0 to 50 in steps of 0.01. The code in Listing 3.4 shows how.

```
% Simulation of a pendulum, for two different
% initial conditions. Mass = 1
```

```

% The pendulum equation: theta'' = - sin(theta)
P = @(t, X) [X(2) ; -sin(X(1))];

% Time:
T=[0:.01:40];
% Initial condition: pi/4
X0=[pi/4; 0];

% Solve
Y = heunvector(P, T, X0);
% Plot both angle Y(1,:) and angular speed Y(2,:)
plot(T, Y)
hold on

% Initial condition: pi/6
X1=[pi/6;0];

% Solve & plot
Y2 = heunvector(P, T, X1);
plot(T, Y2)

```

LISTING 3.4. Simulation of a pendulum with Heun's method.

Exercise 17: Compute approximate solutions to the same systems as in Example 9 using `eulervector` and compare. Which seems more apt? Why?

Exercise 18: Consider the pendulum of Example 9 but with friction. Assume friction is proportional to the angular speed (and with opposite direction), with proportionality constant 0.1. The equation needs to be rewritten taking into account the mass at the end of the pendulum. Plot the graph and compare with the previous exercise (use $l = 1$ everywhere and the mass of your choice). The effect is called “damping,” in this case “exponential damping” because the damping term (what makes the sine waves diminish in amplitude) is exponential.

Exercise 19: Harmonic motion is defined by the second order differential equation

$$\ddot{x} = -\frac{k}{m}x$$

for some elasticity constant k and mass m of the moving body. It is *damped* if there is a damping term which goes against motion:

$$\ddot{x} = -\frac{k}{m}x - r\dot{x}$$

for some damping constant $r > 0$. Study the behaviour of the oscillator with and without damping, using both `eulervector` and `heunvector`. What happens with `eulervector`?

What happens if you set r to some negative value? —

Exercise 20: The ballistic equation describes the motion of a body under gravity. If $(x(t), y(t))$ is its position vector, then the equation is

$$\begin{aligned}\ddot{x} &= 0 \\ \ddot{y} &= -g.\end{aligned}$$

Given an initial position $(0, 0)$ and speed $(1, 1)$, plot the $(x(t), y(t))$ coordinates of the corresponding motion for t from 0 to 20 in steps of 0.1. Use `heunvector`.

What is the differential equation if there is friction and it is proportional to the velocity (but in opposite direction)? Plot the trajectory corresponding to this motion for the same initial conditions as before.

Finally, compute and plot the trajectory if friction is proportional to *the square* of each component of the velocity, in each component. —

CHAPTER 4

Numerical Integration

Unlike in the theory classes, we are not going to study numerical differentiation. However, the student should be able to understand and implement algorithms using both right- and left-sided methods and the symmetric ones (at least for the first and second order derivatives). It is possible that one informal practical be used to show examples of the three methods, for solving ordinary differential equations.

This is one of the simplest practicals, as the topic —numerical integration— will be covered quickly and only the simplest formulas will be implemented. All of them are assumed to be known (because they will have been explained in the theory classes).

1. The simple quadrature formulas

Numerical integration (or *quadrature*, the classical term) is first dealt with as the problem of finding a suitable formula for the whole integration interval. This gives rise to the three most known methods: the midpoint rule, the trapeze rule and Simpson's rule. Start with a function $f : [a, b] \rightarrow \mathbb{R}$. The following are the three basic quadrature formulas:

- (1) The *midpoint rule* uses the value of f at the midpoint $\frac{a+b}{2}$:

$$\int_a^b f(x) dx \simeq (b - a) f\left(\frac{a + b}{2}\right).$$

- (2) The *trapeze rule* uses the value of f at both endpoints:

$$\int_a^b f(x) dx \simeq (b - a) \frac{(f(a) + f(b))}{2}.$$

We prefer stating it like this to emphasize that the area is computed as the width of the interval $(b - a)$ times the mean value of f at the endpoints.

- (3) *Simpson's rule* uses three values: at both endpoints and at the midpoint:

$$\int_a^b f(x) dx \simeq (b - a) \frac{(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b))}{6}.$$

Notice that the denominator 6 is the sum of the coefficients $1 + 4 + 1$ at each point.

Example 10 A possible implementation of the midpoint rule, which receives three parameters, **a**, **b** and **f** (the last one an anonymous function) is included in listing 4.1. In order for it to define a proper function, the corresponding file should be called `midpoint.m`.

```
% Numerical quadrature using the midpoint rule.
% Input: a, b, f, where
%   a, b are real numbers (the endpoints of the integration interval)
%   f   is an anonymous function

function [v] = midpoint(a, b, f)
    v = (b-a).*f((a+b)./2);
end
```

LISTING 4.1. Implementation of the midpoint rule.

A usage example could be:

```
> f = @(x) cos(x);
> midpoint(0, pi, f)
ans = 0
```

Exercise 21: Implement the trapeze rule in an m-file called `trapeze.m`. Use it to compute approximations to the following:

- The integral of $\tan(x)$ from $x = 0$ to $x = \pi/2$.
- The integral of e^x from $x = -1$ to $x = 1$.
- The integral of $\sin(x)$ from $x = 0$ to $x = \pi$.
- The integral of $\cos(x) + \sin(x)$ from $x = 0$ to $x = \pi$.
- The integral of $x^2 + 2x + 1$ from $x = -2$ to $x = 3$.
- The integral of x^3 from $x = 2$ to $x = 6$.

Use also the function `midpoint` as defined in example 10 to compute the same integrals. Use Matlab's `int` function to compute the exact values (or WolframAlpha or your own ability) and compare the accuracy of both methods.

Exercise 22: Implement Simpson's rule in a file called `simpson.m` and use it to compute approximations to the integrals in Exercise 21. Compare the accuracy of this method to that of the other two. Which

is best? Why? Does it always give the best approximation? —

As the reader will have noticed, the above are rules which use a formula applied to one, two or three points in the interval and approximate the value of the integral on the whole stretch $[a, b]$. One can get more precise results dividing $[a, b]$ into a number of subintervals and applying each formula to each of these. This way one gets the *composite* quadrature formulas.

2. Composite rules

Of course, dividing the interval requires somehow knowing how many subintervals are needed. We shall assume the user provides this number as an input. Hence, the functions we shall implement will receive *four* parameters: the endpoints, the function and another one, the number of subintervals.

Example 11 For instance, a possible implementation of the *composite midpoint rule* can be read in listing 4.2. Notice how the `sum` operation adds all the values of a vector (in this case the vector $\mathbf{f}(\mathbf{m})$ of values of \mathbf{f} at the midpoints).

```
% Numerical quadrature using the composite midpoint rule.
% Input: a, b, f, n=3, where
%   a, b are real numbers (the endpoints of the integration interval)
%   f   is an anonymous function
%   n   is the number of subintervals in which to divide [a,b]

function [v] = composite_midpoint(a, b, f, n)
    l = (b-a)./n;
    % midpoints
    a1 = a + l./2;
    b1 = b - l./2;

    % Can you explain why this is correct?
    m = linspace(a1, b1, n);
    v = 1.*sum(f(m));
end
```

LISTING 4.2. An implementation of the composite midpoint rule. —

Exercise 23: Use the code of `composite_midpoint` to compute approximations to the integrals in Exercise 21, with different values for the parameter n . Compare the results with the ones obtained before.

—

Exercise 24: Implement the composite trapeze and Simpson's rule (in two different files called, respectively, `composite_trapeze.m` and `composite_simpson.m`). Use these implementations to compute approximations to the integrals in Exercise 21. Compare the results. —

Exercise 25: If one uses the composite trapeze rule with $2n$ subintervals and Simpson's rule with n (for example, setting `n` to 6 for the trapeze rule and to 3 for Simpson's), one is using the same evaluation points (verify this). Are the approximations obtained equally good? Which is better? Why do you think this happens? —

CHAPTER 5

Interpolation

After a brief digression on integration (which may be useful for computing areas related to solutions of ODEs) we return to the problem of finding an approximate “solution” to an ODE. All of the methods explained in Chapter 2 gave numerical approximations to the solution of an ODE as a list of values at the points of a network on the x -axis. However, in many instances, values of the solution at points not on the network will be required (for example, to plot the solution or to approximate the values at points not on the network). When the values required fall *inside* the network, this is known as the *interpolation* problem. If the values required fall *outside* it, one is *extrapolating*. We shall mostly deal with the first problem. The second one is hard to tackle and requires some extra knowledge of the function..

We shall only explain *one-dimensional* interpolation. Techniques for more than one dimension obviously exist but they are all related to the ones we shall explain: a good command of one-dimensional tools permits an easy grasp of the higher dimensional ones.

Assume a vector $\mathbf{x} = (x_1, \dots, x_n)$ of (ordered) x -coordinates is given and another one $\mathbf{y} = (y_1, \dots, y_n)$ of the same length represents the values of a function f at each point of \mathbf{x} . The problem under consideration consists in approximating the values of f at any point between x_1 and x_n .

1. Linear interpolation

One of the simplest solutions to the interpolation problem is to draw line segments between each (x_i, y_i) and (x_{i+1}, y_{i+1}) for each i and, if $\tilde{x} \in [x_i, x_{i+1}]$, approximate the value of $f(\tilde{x})$ as the corresponding line in the segment. This means using the approximation

$$f(\tilde{x}) \simeq \frac{y_{i+1} - y_i}{x_{i+1} - x_i}(\tilde{x} - x_i) + y_i,$$

having previously found the i such that $\tilde{x} \in [x_i, x_{i+1}]$. Notice that *there may be two such i* , but the result is the same in either case (why?).

Exercise 26: Implement the above interpolation method. Specifically, write a file called `linear_int.m` implementing a function `linear_int`

which, given two vectors \mathbf{x} and \mathbf{y} (corresponding to the \mathbf{x} and \mathbf{y} above) and a value x_1 , returns the value of the linear interpolation at x_1 corresponding to \mathbf{x} and \mathbf{y} . A couple of examples of calling could be

```
> x = [1 2 3 4 5];
> y = [0 2 4.1 6.3 8.7];
> linear_int(x, y, 2.3)
ans = 2.6300
> linear_int(x, y, [2.3 3.5])
ans =
    2.6300    5.6400
```

Notice how the last parameter can be a vector of values on which to compute the interpolation. —

Exercise 27: Use the function defined in Exercise 26 to plot the graph of the linear interpolation corresponding to the following clouds of points:

- The points $(1, 2), (2, 3.5), (3, 4.7), (4, 5), (5, 7.2)$.
- The points $(-2, 3), (-1, 3.1), (0, 2.8), (1, 3.5), (2, 4), (3, 5.7)$.
- On the x -axis, a list of 100 points evenly distributed between 0 and π . On the y -axis, the values of the function $\sin(100x)$ at those points.

2. Cubic splines

Linear interpolation is useful mainly due to its simplicity. However, in most situations, the functions with which one is working are differentiable (and in many cases, several times so), and linear interpolation does not usually satisfy this condition. To solve this problem, splines were devised.

As was explained in the theory classes, *cubic splines* are the most used and they are the ones we shall implement.

Before proceeding, we shall explain the internal commands Matlab uses for dealing with cubic splines. Then we shall implement a spline function in detail.

2.1. The spline and ppval functions. Given a cloud of points described by the vectors of x - and y -coordinates, say $\mathbf{x} = (x_1, \dots, x_n)$ and $\mathbf{y} = (y_1, \dots, y_n)$, Matlab can compute different types of splines. The command for doing so is `spline` and it takes as input, in its most basic form, two vectors, \mathbf{x} and \mathbf{y} . Then:

- If x and y are of the same length, then `spline(x,y)` returns the *not-a-knot* cubic spline interpolating the cloud given by x and y .
- If y has exactly *two* more components than x , then the call `spline(x,y)` returns the interpolating cubic spline using the cloud given by x and $y(2:\text{end}-1)$ with the condition that the derivative of the spline at the first point is $y(1)$ and the derivative at the last point is $y(\text{end})$. These splines are called, for obvious reasons, *clamped*.

Example 12 Let us work with the first cloud of points in Exercise 26. In this case, x is `[1 2 3 4 5]` whereas y is `[2 3.5 4.7 5 7.2]`.

- The *not-a-knot* spline is computed straightaway using `spline`:

```
> x = [1 2 3 4 5];
> y = [2 3.5 4.7 5 7.2];
> p = spline(x, y);
```

The object returned by `spline`, which we have called `p`, has a special nature: it is a *piecewise polynomial*. In order to evaluate it, one has to use the function `ppval`:

```
> ppval(p, 2.5)
ans = 4.2281
```

which can be used with vectors as well:

```
> ppval(p, [1:.33:2.33])
ans =
```

```
    2.0000    2.4389    2.9510    3.4841    3.9861
```

and hence, can be used for plotting the actual spline:

```
> u = linspace(1, 5, 300);
> plot(u, ppval(p,u));
```

- The *clamped* cubic spline imposes specific values for the first derivative at the endpoints. In order to compute it using Matlab one has to add this condition as the first and last values of the y parameter. For the same cloud of points as above, and setting the first derivative at the endpoints to 0, one would write:

```
> x = [1 2 3 4 5];
> y = [2 3.5 4.7 5 7.2];
> q = spline(x, [0 y 0]);
```

Let us plot `q` on the same graph as `p`:

```
> hold on
> plot(u, ppval(q, u), 'r');
```

What is the difference?

- Take into account that Matlab does not include in its default toolbox the ability to compute *natural* splines (those for which the second derivative at the endpoints is 0). We shall implement this below.

Exercise 28: Describe (in detail) at least two situations in which linear interpolation should be preferred to cubic splines. Same for the reverse.

Can you come up with examples in which the natural spline is better suited than the “not-a-knot”? What about the reverse?

2.2. Implementing the natural cubic spline. We shall write a long function implementing the natural cubic spline. From the theory, we know that solving a linear system of equations is required. However, this system, once the problem is stated properly, has a very simple structure. We shall solve it using Matlab’s solver. Also, the function shall return a piecewise defined polynomial, using the `mkpp` utility. This section should be read more as a thorough exercise on Matlab programming than as a useful example.

From polynomials to a linear system. The problem under consideration consists in, given $\mathbf{x} = (x_1, \dots, x_n)$ and $\mathbf{y} = (y_1, \dots, y_n)$, find polynomials $P_1(x), \dots, P_{n-1}(x)$ (notice that there are $n - 1$ polynomials, not n) satisfying the following conditions:

- (1) Each P_i passes through the points (x_i, y_i) and (x_{i+1}, y_{i+1}) .
- (2) At each x_i , for $i = 2, \dots, n - 1$, the derivative of $P_i(x)$ equals that of $P_{i-1}(x)$.
- (3) At each x_i , for $i = 2, \dots, n - 1$, the second derivative of P_i equals that of $P_{i-1}(x)$.

It is easy to check that those conditions give a total of $4(n - 1) - 2$ linear equations for the coefficients of the polynomials $P_i(x)$. As there are $4(n - 1)$ coefficients, there are two missing equations for a system with a unique solution. These two equations allow for the different types of splines (natural, not-a-knot, etc.).

Let us express each polynomial $P_i(x)$ (for $i = 2, \dots, n-1$) as

$$P_i(x) = a_i + b_i(x - x_{i-1}) + c_i(x - x_{i-1})^2 + d_i(x - x_{i-1})^3$$

and each difference

$$x_i - x_{i-1} = h_i$$

¹This is a bit awkward but simplifies the notation.

(for $i = 2, \dots, n$ also). After some algebraic manipulations (which can be found in the theory or anywhere on the Internet), one arrives at the following set of equations:

$$h_{i-1}c_{i-1} + (2h_{i-1} + 2h_i)c_i + h_i c_{i+1} = 3 \left(\frac{y_i - y_{i-1}}{h_i} - \frac{y_{i-1} - y_{i-2}}{h_{i-1}} \right)$$

for $i = 3, \dots, n-1$ (this gives $n-1-3+1 = n-3$ equations, two less than $n-1$, exactly as expected). There are also explicit linear expressions for each a_i , b_i and d_i in terms of the c_i . One can easily check that the equations are independent. Those $n-3$ equations can be written as a linear system $Ac = \alpha$, where A is the $(n-3) \times (n-1)$ matrix

$$A = \begin{pmatrix} h_2 & 2(h_2 + h_3) & h_3 & 0 & \dots & 0 & 0 & 0 \\ 0 & h_3 & 2(h_3 + h_4) & h_4 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & h_{n-2} & 2(h_{n-2} + h_{n-1}) & h_{n-1} \end{pmatrix}$$

and c is the vector column of the unknowns $(c_2, \dots, c_n)^t$, whereas α is

$$\begin{pmatrix} \alpha_3 \\ \alpha_4 \\ \vdots \\ \alpha_{n-1} \end{pmatrix}$$

and each α_i (for $i = 3, \dots, n-1$) is

$$(2) \quad \left(\alpha_i = 3 \left(\frac{y_i - y_{i-1}}{h_i} - \frac{y_{i-1} - y_{i-2}}{h_{i-1}} \right) \right)$$

There are obviously two missing equations for a square system. As we want to implement the *natural spline* condition, which reads as $d_2 = 0$ and $d_n = 0$, we need to “translate” these conditions into new equations involving only the c_i coefficients. After some algebraic manipulations, one obtains the following:

$$c_2 = 0, \\ \frac{h_{n-1}}{2}c_{n-1} + (h_n + h_{n-1})c_n = \frac{3}{2} \left(-\frac{y_{n-1} - y_{n-2}}{h_{n-1}} + \frac{y_n - y_{n-1}}{h_n} \right)$$

Letting $\alpha_n = 6(y_n - y_{n-1})/h_n$ and $\alpha_2 = 0$, the complete system is

$$\tilde{A}c^t = \tilde{\alpha}$$

where \tilde{A} is A together with a first row

$$(1 \ 0 \ 0 \dots \ 0)$$

and a last row

$$(0 \ 0 \ \dots \ 0 \ 2h_{n-1} \ (4h_{n-1} + 5h_n))$$

and $\tilde{\alpha}$ is the same α with $\alpha_2 = 0$ on the first row and $\alpha_n = 6(y_n - y_{n-1})/h_n$.

Writing the system as a Matlab matrix: so far, we have just written “in human terms” the rows of the linear system to be solved, and the column corresponding to the independent terms. We now proceed to write the appropriate Matlab code which describes it.

Before proceeding any further, we know that $a_i = y_{i-1}$ for $i = 2, \dots, n$, which we can write (recalling that indices in Matlab start at 1, whereas our polynomials start at 2):

```
a = y(1:end-1);
```

For the system of equations, we need a matrix, which we shall call A of size $(n-1) \times (n-1)$ (remember that there are $n-1$ polynomials, not n). We know that the first row is a 1 followed by $n-2$ zeros, the following $n-3$ rows are those of A in Equation 2.2 and the last one is a list of $n-3$ zeros followed by $2h_{n-1}, (4h_{n-1} + 5h_n)$. The matrix A is tridiagonal, and its structure can be described as:

- The line below the main diagonal is $(h_2, h_3, \dots, h_{n-1})$.
- The line above the main diagonal is (h_3, h_4, \dots, h_n) .
- The diagonal is twice the sum of the lines above.

So it turns out that the vector (h_2, h_3, \dots, h_n) will be useful. Recall that $h_i = x_i - x_{i-1}$. If \mathbf{x} is the Matlab vector corresponding to the x -coordinates, then setting

```
> h = diff(x);
```

makes \mathbf{h} the vector of differences, the one we wish to use. (Notice that h_2 is *the first coordinate of \mathbf{h}* because $h_2 = x_2 - x_1$).

The command to construct matrices with diagonal values is `diag`. It works as follows:

```
diag(v, k)
```

will create a *square* matrix whose \mathbf{k} -th diagonal contains the vector \mathbf{v} . If \mathbf{k} is missing, it is set to 0. The \mathbf{k} -th diagonal is the diagonal row which is \mathbf{k} -steps away from the main diagonal. Thus, the 0-th diagonal is the main one, the 1-diagonal is the one just to the right of the main one, the -1-diagonal is the one just to the left, etc. For example,

```
> diag([2 -1 3], -2)
```

```
ans =
```

```
0 0 0 0 0
```



```

0  0  0  0  0
2  0  0  0  0
0 -1  0  0  0
0  0  3  0  0

```

This allows for a very fast specification of the matrix \tilde{A} in of the linear system to be solved. The -1 -diagonal is

```
[h(1:end-1) h(end-1)/2]
```

the 1 -diagonal is

```
[0 h(2:end)]
```

and the proper diagonal is

```
[1 2*(h(1:end-1) + h(2:end)) h(end)+h(end-1)]
```

where `end` means, in Matlab, the last index of a vector. From these values, it should be easy to understand that the matrix A of the linear system to be solved (which is \tilde{A} above) can be defined as follows (we divide the matrix into three “diagonal” ones for clarity):

```

A1 = diag([h(1:end-2) h(end-1)/2], -1);
A2 = diag([0 h(2:end-1)], 1);
A3 = diag([1 2*(h(1:end-2)+h(2:end-1)) h(end)+h(end-1)]);
A  = A1+A2+A3;

```

The column vector $\tilde{\alpha}$ is defined as follows: recall that the first element is 0, the next $n - 3$ are as in Equation 2 and the last one is $6(y_n - y_{n-1})/h_n$. Thus, letting `dy = diff(y)`, we can write (notice the dots before the slashes in the second element):

```

alpha = [0 3*(dy(2:end-1)./h(2:end-1)-dy(1:end-2))./h(1:end-2))
         3/2*(-dy(end-1)/h(end-1)+dy(end)/h(end))]' ;

```

Once the system of equations $\tilde{A}c' = \tilde{\alpha}'$ has been properly set up, one solves it using Matlab’s solver:

```
c = (A\alpha)';
```

The translation of the explicit expressions for b and d (in the theory), into matlab goes as (where `n` is the number of interpolation polynomials):

```

% Initialize b and d
b = zeros(1,n);
d = zeros(1,n);

% unroll all the coefficients as in the theory
k = 1;
while(k<n)
    b(k) = (y(k+1)-y(k))/h(k) - h(k) *(c(k+1)+2*c(k))/3;
    k=k+1;

```

```

end
d(1:end-1) = diff(c)./(3*h(1:end-1));

% the last b and d have explicit expressions:
b(n) = b(n-1) + h(n-1)*(c(n)+c(n-1));
d(n) = (y(n+1)-y(n)-b(n)*h(n)-c(n)*h(n)^2)/h(n)^3;

```

At this point, we have computed all the coefficients of the interpolating polynomials. The way to define a piecewise-polynomial function in matlab is using `mkpp`, which takes as arguments the vector of x -coordinates defining the intervals on which each polynomial is used (in our case the same as the \mathbf{x} input vector) and a matrix containing the coefficients of each polynomial *relative to* x_{i-1} , that is, in our case²:

$$\begin{pmatrix} d_n & c_n & b_n & a_n \\ d_{n-1} & c_{n-1} & b_{n-1} & a_{n-1} \\ \vdots & & & \\ d_2 & c_2 & b_2 & a_2 \end{pmatrix}$$

which gives, in matlab:

```
f = mkpp(x, [d; c; b; a]');
```

Putting everything together in a file called `natural_spline.m`, we get the code of Listing 5.1.

```

% natural cubic spline: second derivative at both
% endpoints is 0. Input is a pair of lists describing
% the cloud of points.
function [f] = natural_spline(x, y)
    n = length(x)-1;

    % variables and coefficients for the linear system,
    % these are the ordinary names. Initialization
    h = diff(x);
    dy = diff(y);
    F = zeros(n);
    a = y(1:end-1);
    alpha = [0 3*(dy(2:end-1)/h(2:end-1)-dy(1:end-2)/h(1:end-2)) 3/2*(-dy(
        end-1)/h(end-1)+dy(end)/h(end))];
    A1 = diag([h(1:end-2) h(end-1)/2], -1);
    A2 = diag([0 h(2:end-1)], 1);
    A3 = diag([1 2*(h(1:end-2)+h(2:end-1)) h(end)+h(end-1)]);
    A = A1+A2+A3;

    % Solve the c coefficients:

```

²Important: remember that Matlab understands a vector `[d c b a]` as a polynomial taking the coefficients from greatest to lowest degree, in the example, $dx^3 + cx^2 + bx + a$.

```

c = (A\alpha)';

% Initialize b and d
b = zeros(1,n);
d = zeros(1,n);

% unroll all the coefficients as in the theory
k = 1;
while(k<n)
    b(k) = (y(k+1)-y(k))/h(k) - h(k) *(c(k+1)+2*c(k))/3;
    k=k+1;
end
d(1:end-1) = diff(c)./(3*h(1:end-1));

% the last b and d have explicit expressions:
b(n) = b(n-1) + h(n-1)*(c(n)+c(n-1));
d(n) = (y(n+1)-y(n)-b(n)*h(n)-c(n)*h(n)^2)/h(n)^3;

% finally, build the piecewise polynomial (a Matlab function)
% we might implement it by hand, though
f = mkpp(x,[d; c; b ;a ]');
end

```

LISTING 5.1. Function implementing the computation of the natural spline for a cloud of points.

Example 13 Consider the points $(0, 1), (1, 3), (2, 7), (4, 3), (6, 0)$. The natural cubic spline passing through them can be computed and plotting, using the function just defined, as follows:

```

> x=[0 1 2 4 6];
> y=[1 3 7 3 0];
> P=natural_spline(x,y);
> u=[0:.01:6];
> plot(u, ppval(P, u));

```

Notice how, instead of $P(u)$, one needs to use the function `ppval` to evaluate a *piecewise defined function*. In order to visually verify that P passes through all the points, one can plot them on top of P :

```

> hold on;
> plot(x,y,'*r');

```

Example 14 A more complicated example: let us try to estimate (visually) the difference between the *sine* function and a natural cubic spline, along the period $[0, 2\pi]$, using 10 points. This could be done as follows:

```

> x=linspace(0, 2*pi, 10);
> y=sin(x);
> Q=natural_spline(x, y);
> u=[0:.01:2*pi];
> clf;
> plot(u, sin(u));
> hold on;
> plot(u, ppval(Q, u), 'r');

```

LISTING 5.2. Comparison of the graphs of the sine function and a cubic spline with 10 points.

The graphs should be indistinguishable. This gives an idea of the power of cubic splines: for *sufficiently well-behaved functions*, they give surprisingly good approximations. —

Exercise 29: Matlab has a `spline` function, as explained above. Compare the plots of this function with those of the natural spline for examples 14 and 13. Which gives a better approximation to the sine function? —

Exercise 30: Let $f(x) = \cos(\exp(x))$, for $x \in [2, 5]$. Let P be the natural spline interpolating the values of f on 10 points from 2 to 5. Compare the plots of P and of f on that interval. Are they similar? Are they different? Why do you think that happens? How do you think you can fix this problem? —

Exercise 31: Let $f(x) = \exp(\cos(x))$, for $x \in [0, 6\pi]$. Let Q be the natural spline interpolating the values of f on 20 points. Plot both f and Q on that interval. Where are the most noticeable differences between those two plots? Can you get a better approximation using the `spline` function? Why? —

Exercise 32: Consider the differential equation

$$y' = \frac{y}{1 + x^2}.$$

Use any of the algorithms defined in Chapter 2 to compute the approximate values of a solution on the interval $[0, 5]$ with initial condition $y(0) = 1$ and using a step of size 0.5. Use a natural spline to interpolate the values of an approximate solution passing through those points.

The true solution to that ODE is $y(x) = ce^{\arctan(x)}$, for c a constant. Compute the constant for the initial value $y(0) = 1$ and compare the plots of the true solution and the spline. Explain the difference between both plots: is it due to the nature of the spline or to the approximate

solution of the ODE? Would the graphs be more similar if more points were used?

Explain as much as possible. This is a very important exercise. Perform different computations, use a different number of intermediate points, etc. . .

3. Least Squares Interpolation

Given a cloud C of N points and a *linear family* V (that is, a *vector space*) of functions which “are supposed to properly represent the cloud of points,” the problem of finding the *best* approximation to the cloud by a function in V can be understood in different ways. The most common is the *least squares approximation*, which was explained in the theory classes and which can be stated as follows:

Let $(x_1, y_1), \dots, (x_N, y_N)$ be the points in C and let $\{f_1, \dots, f_n\}$ be a basis of V . The *least squares interpolation problem* for C and V consists in finding coefficients a_1, \dots, a_n such that the number

$$E(a_1, \dots, a_n) = \sum_{i=1}^N (a_1 f_1(x_i) + \dots + a_n f_n(x_i) - y_i)^2$$

is minimal. That number (which depends, obviously, on the coefficients) is called the *total quadratic error*.

There are several ways to solve this problem. However, the one we explain in theory translates the differential problem (finding a global minimum) into a system of n linear equations (as many equations as *functions in the basis*). The details have been explained in the theory classes. In the end, the coefficients a_1, \dots, a_n are the solution to the following linear system:

$$(3) \quad XX^t \begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix} = X \begin{pmatrix} y_1 \\ \vdots \\ y_N \end{pmatrix}$$

(notice that both sides evaluate to column vectors of n components). The matrix X is

$$X = \begin{pmatrix} f_1(x_1) & f_1(x_2) & \dots & f_1(x_N) \\ \vdots & \vdots & \ddots & \vdots \\ f_n(x_1) & f_n(x_2) & \dots & f_n(x_N) \end{pmatrix}$$

and X^t is its transpose. Equation (3) is always (if $N > n$, which is almost always an obvious requirement) a compatible system (which might have non-unique solution in some cases) but it is usually *ill-posed*.

Thus, the least squares interpolation problem requires a set of N points (the cloud to be interpolated) and a family of n functions (with $n < N$, as a matter of fact, N is usually large and n small). With these data, the problem is just a linear system of equations.

Instead of having to compute the matrix X by hand each time a least-squares interpolation problem arises, we can define a function which, given the cloud of points and a family of n functions, finds the coefficients a_1, \dots, a_n .

Example 15 The simplest useful example is the interpolation of a cloud of points by a linear function. Linear functions have always the form $a + bx$, so that the vector space they span has two generators $f_1(x) = 1$ and $f_2(x) = x$ (there are two values to compute, a and b). Let the cloud be $(1, 2), (2, 2.1), (3, 2.15), (4, 2.41), (5, 2.6)$. It is easy to guess that the interpolating line will be approximately $y = 0.1(x - 1) + 2 = 1.9 + 0.1x$ (the slope is approximately 0.1 and the line passes more or less through the point $(1, 2)$). Let us solve this interpolation problem using Matlab:

```
> x=[1 2 3 4 5];
> y=[2 2.1 2.15 2.41 2.6];
> f1=@(x) 1 + x.*0;
> f2=@(x) x;
> X=[f1(x); f2(x)]
X =

    1    1    1    1    1
    1    2    3    4    5

> A=X*X'
A =

    5    15
   15    55

> Y=X*y'
Y =

   11.260
   35.290

> coefs=A\Y
coefs =

    1.79900
    0.15100
```

X	Y
1.0	6.20
1.5	9.99
2.0	15.01
2.5	23.23
3.0	32.70
3.5	43.08
4.0	54.01
4.5	65.96
5.0	80.90

TABLE 1. Data following a quadratic formula.

That is, the linear function which interpolates the cloud of points by least squares is $y = 1.799 + 0.151x$, which resembles our guess. —

REMARK. Notice that there is a little issue: when defining a function which returns a constant value $f_1(x) = 1$, one has to “make it vectorial” by adding some null vector (in this case, $0.*x$). Otherwise, the function would return a number, not a vector (but we need a vector, with as many components as x).

When doing least squares interpolation, one usually has a much larger cloud of points (dozens or even hundreds or thousands of points). This forces one to read the data from a file (entering them by hand is error-prone and too slow). This can be done using different commands. The simplest one is `load`, but we are not going to enter into details. Use the documentation of Matlab if you are interested.

Example 16 The data in Table 1 comes from an experiment. It is known that variable Y depends quadratically on variable X (that is, there is a quadratic formula which relates X to Y). Using least-squares interpolation, find the most adequate coefficients for the formula.

As we know that Y depends quadratically on X , the vector space of functions we are dealing with is spanned by $1, x, x^2$. Let $f_1 = 1, f_2 = x, f_3 = x^2$. We could use Matlab as follows (omitting the output where it is irrelevant):

```
> x=[1:.5:5];
> y=[6.2 9.99 15.01 23.23 32.7 43.08 54.01 65.96 80.9];
> % define the functions:
> f1=@(x) 1+0.*x;
> f2=@(x) x;
> f3=@(x) x.^2;
> % main matrix
```

```

> X=[f1(x); f2(x); f3(x)];
> A=X*X';
> Y=X*y';
> % the system is A*a'=Y, use matlab to solve it:
> A\Y
ans =

    0.55352
    2.27269
    2.75766
> % this means that the least squares interpolating
> % polynomial of degree 2 is
> % 0.55352 + 2.27269*x + 2.75766*x.^2
> % plot both the cloud of points and the polynomial
> plot(x,y);
> hold on
> u=[1:.01:5];
> plot(u, 0.55352 + 2.27269.*u + 2.75766*u.^2, 'r')

```

Notice how the least squares interpolating polynomial does not pass through all the points in the cloud (it may even pass through none).

—

REMARK 2. The least squares interpolation problem needs not be only about finding *polynomials* which fit a cloud of points. Depending on the *linear model*, one may have exponential functions, trigonometric functions, and many other. However, notice that the model must be *linear* in order to allow the use of least-squares. Otherwise, one will most likely run into trouble.

Exercise 33: A new theoretical development has shown that the data in Table 1 is best described by a cubic function. Use least squares interpolation to compute the best-fitting cubic function. Are the coefficients of degrees 0, 1 and 2 similar to those of Example 16? Does the cubic polynomial resemble the data better or worse than the quadratic one?

This exercise is interesting because fitting a curve with a polynomial is, in most cases, something inadvisable, unless the polynomial is of degree 1 (i.e. a straight line). One should have very strong arguments in favor of using a polynomial of degree greater than one to fit a curve (there are some physical laws in which degree 2 and 3 polynomials appear, however. Give some examples).

Exercise 34: Table 2 represents data from an experiment. It is known that the real data follows a function of the form $y(x) = a \log(x) + bx + ce^x$, for some a, b and c . Using linear least-squares interpolation,

X	Y
2.0	11.39
2.7	15.31
3.4	18.18
4.1	19.80
4.8	19.76
5.5	15.03
6.2	1.74
6.9	-29.67

TABLE 2. Data following a log-lin-exp formula.

X	Y	X	Y	X	Y	X	Y
10	8.04	10	9.14	10	7.46	8	6.58
8	6.95	8	8.14	8	6.77	8	5.76
13	7.58	13	8.74	13	12.74	8	7.71
9	8.81	9	8.77	9	7.11	8	8.84
11	8.33	11	9.26	11	7.81	8	8.47
14	9.96	14	8.10	14	8.84	8	7.04
6	7.24	6	6.13	6	6.08	8	5.25
4	4.26	4	3.10	4	5.39	19	12.50
12	10.84	12	9.13	12	8.15	8	5.56
7	4.82	7	7.26	7	6.42	8	7.91
5	5.68	5	4.74	5	5.73	8	6.89

TABLE 3. Anscombe's quartet.

find a, b and c . Can you think of a physical, social or biological phenomenon following a law of that type? —

Exercise 35: Table 3 contains what is called *Anscombe's quartet*. The four lists are remarkable for several statistical properties they share. We are just going to focus on the linear least squares interpolating line $a + bx$. For each of the lists, find the best fit. After finding the fit, plot each of the interpolating lines and each cloud of points *on the same graph*.

What can you infer from your results? —

Exercise 36: An experiment computes the value of kinetic energy from the velocity of a moving object. Table 4 shows the results of five runs of it with the same object. Give a reasonable value for the mass of the object from the data. Velocity is given in m/s while E is in Joules.

v	E
1.0	8.05
1.5	16.97
2.3	39.69
2.7	55.58
3.0	66.91

TABLE 4. Kinetic energy against velocity, output of an experiment.

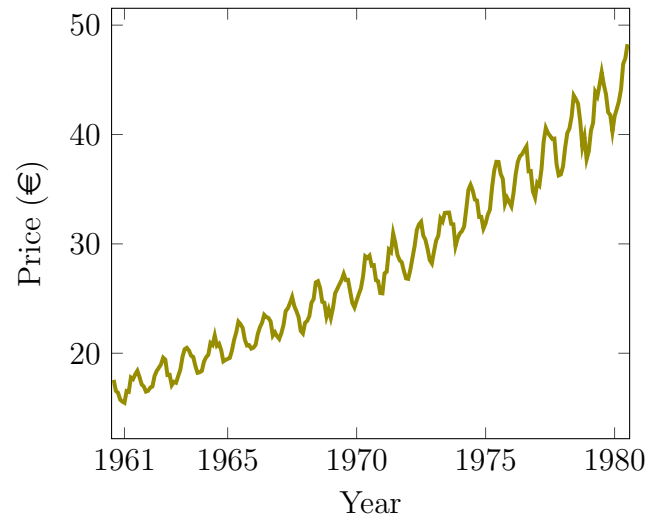
Exercise 37: Table 5 is the outcome of an experiment which consists in computing the distance traveled by an object after some time. It is known that the object moves with uniform acceleration and that it always starts with the same initial velocity. Give reasonable values for the acceleration and the initial speed.

t (s)	d (m)
1	4.89
2	11.36
3	21.64
4	34.10
5	50.05
6	69.51

TABLE 5. Distance against time for an experiment.

Exercise 38: The following plot shows the mean price of a can of beer (in pesetas) at each month from 1960 to 1980 (there are 240 values in the series). The plot has two remarkable properties: on the one hand, it “seems to increase with time,” on the other, the price has a wavy behavior (with local lows and highs at intervals of the same length). As a matter of fact, minima and maxima happen (approximately) with a year of difference. This behavior is called “seasonality” (the season of the year affects the price of commodities: in this case, beer is drunk more frequently in the Summer, as people are more thirsty, and prices tend to be higher than in the Winter). How would you model this graph in order to find a “suitable” fitting function?

One has to be aware that *prices are always modeled with products, not with additions*: prices increase by a rate, not by a fixed amount (things are “more or less expensive” in rate, you would not complain of a computer costing €5 more but you would if the loaf of bread had



that same increase in price). So, the relevant data should not be that in the graph but its logarithm (which increases and decreases in absolute values, not relative ones). Hence, one should aim at a least-squares interpolation of the logarithm of the true values³.

Once logarithms are taken, the curve should be fit (interpolated) as a linear function with a seasonal (yearly) modification: how would you perform this interpolation? (One needs to use the sine and cosine function, but how?). What functions would you use as basis? Why?

The data can be found at <http://pfortuny.net/prices.dat>. —

³Despite not being completely correct, the fact that the values are much larger than 0 makes using logarithms and fitting the new curve with least squares useful, albeit inexact.

CHAPTER 6

Linear Systems of Equations

In the Theory classes several algorithms for solving linear systems of equations have been explained. Their interest is not only for problems which can, by themselves, be expressed as linear equations (like in Statics, for example) but, as the student should have already realized, for solving problems which appear as intermediate steps of others: mainly, spline interpolation (recall that in order to compute the cubic spline, solving a tridiagonal system is required), linear least-squares interpolation. Although we have not covered this in the course, any attempt to solve a differential equation using implicit methods or the numerical solution of partial differential equations lead to linear systems of equations, usually of huge size (think of thousands and millions of rows).

As the student knows, one can divide the methods of solving linear systems of equations into two main groups: “exact” methods (those who aim to produce an *exact* solution) and “iterative” methods, which use a fixed-point iteration to approximate the solution step by step.

1. Exact methods

The first exact method, and the most useful for small systems (up to 4 or 5 equations) is Gauss’s reduction algorithm. This algorithm will serve us to show some programming techniques in Matlab. One should be able to reproduce the program from the description of the algorithm given in the Theory Lecture Notes. We include it for the sake of completeness in Algorithm 4

Example 17 — Implementation of Gauss’s reduction method in Matlab. The code in Listing 6.1 includes an implementation of Gauss’s reduction method in Matlab.

```
function [L, At, bt] = gauss(A,b)
    n = size(A);
    m = size(b);
    if(n(2) ~= m(1))
        warning('The sizes of A and b do not match');
        return;
    end
```

Algorithm 4 Gauss' Algorithm for linear systems

Input: A square matrix A and a vector b , of order n

Output: Either an error message or a matrix \tilde{A} and a vector \tilde{b} such that \tilde{A} is upper triangular and the system $\tilde{A}x = \tilde{b}$ has the same solutions as $Ax = b$

★START

$\tilde{A} \leftarrow A, \tilde{b} \leftarrow b, i \leftarrow 1$

while $i < n$ **do**

if $\tilde{A}_{ii} = 0$ **then**

return ERROR [division by zero]

end if

 [combine rows underneath i with row i]

$j \leftarrow i + 1$

while $j \leq n$ **do**

$m_{ji} \leftarrow \tilde{A}_{ji} / \tilde{A}_{ii}$

 [Next line is usually a loop, careful here]

$\tilde{A}_j \leftarrow \tilde{A}_j - m_{ji} \tilde{A}_i$ [*]

$\tilde{b}_j \leftarrow \tilde{b}_j - m_{ji} \tilde{b}_i$

$j \leftarrow j + 1$

end while

$i \leftarrow i + 1$

end while

return \tilde{A}, \tilde{b}

```

At=A; bt=b; L=eye(n);
k=1;
while (k<n(1))
    l=k+1;
    if(At(k,k) == 0)
        warning('There is a 0 on the diagonal');
        return;
    end
    % careful with rows & columns:
    % L(1,k) means ROW 1, COLUMN k
    while(l<=n)
        L(1,k)=At(1,k)/At(k,k);
        % Combining rows is easy in Matlab
        At(1,k:n) = [0 At(1,k+1:n) - L(1,k) * At(k,k+1:n)];
        bt(1)=bt(1)-bt(k)*L(1,k);
        l=l+1;
    end
    k=k+1;
end
end

```

end

LISTING 6.1. Gauss's method in Matlab.

There are some relevant remarks to be made about the code. To begin with, notice that `i` and `j` are *reserved names* for Matlab (they both represent the complex root of unity), so our code, instead of `i` and `j`, uses `k` and `l`.

- First of all, the size of the coefficient matrix and the independent terms are checked for equality. If they are not, a warning is shown and the program finishes. The variable `n` is set to the number of rows of `A`.
- Then the output variables are initialized (this must be done at the beginning to prevent strange errors): they are `At`, which holds the final upper triangular matrix, `bt`, which holds the final independent terms and `L`, which is the transformation matrix.
- Then the main `while` loop starts (which iterates on the rows of `At`), using the variable `k`, starting at `k=1` until `k>=n(1)` (which means, until the last-but-one row). Then, for each row:
 - The corresponding element `At(k,k)` is verified to be nonzero. If it is zero, the algorithm finishes (there is no pivoting done in this program) with an error message.
 - If the diagonal element is nonzero, then for each row under the `k`-th one (this is the `while l<=n`),
 - * The corresponding multiplier is computed and stored in `L(1,k)`: $At(1,k)/At(k,k)$.
 - * The `l`-th row of `At` is updated. The line

$$At(1,k:n) = [0 \quad At(1,k+1:n) - L(1,k)*At(k,k+1:n)];$$
 does the following: the element `At(1,k)` is set to 0. Then the elements `At(1, r)` are set to the corresponding value of the combination $\tilde{A}_j \leftarrow \tilde{A}_j - m_{ji}\tilde{A}_i$. Notice that the elements to the left of `k` are already zero. Why?
 - * Then the independent terms are updated accordingly.
 - * The counter `l` is increased by one.
 - The counter `k` is increased by one.
- There is no ending statement because all the output variables have already been computed at this point.

The function thus defined, `gauss`, returns three values: the lower triangular matrix `L` (which contains the multipliers), the upper triangular

matrix $\mathbf{A}t$, which is the *reduced* matrix and the new vector of independent terms $\mathbf{b}t$. In order to solve the initial system $Ax = b$, one needs only solve the new one $\tilde{A}x = \tilde{b}$ (where \tilde{A} is $\mathbf{A}t$ and \tilde{b} is $\mathbf{b}t$). —

Exercise 39: Use the function `gauss` defined in listing 6.1 (which implies your saving it in an adequate file in an adequate directory) to compute the reduced form of the following systems of equations: —

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 33 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 2 \\ 7 \\ 8 \end{pmatrix} \quad \begin{pmatrix} 2 & -1 & 3 \\ 6 & -3 & 2 \\ 0 & 2 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} -1 \\ -1 \\ -1 \end{pmatrix}$$

$$\begin{pmatrix} -1 & 2 & 1 & 3 \\ 2 & -4 & 4 & 1 \\ 0 & 1 & 2 & 3 \\ -1 & -2 & -3 & -4 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ t \end{pmatrix} = \begin{pmatrix} -4 \\ 3 \\ -2 \\ 1 \end{pmatrix} \quad \begin{aligned} 2x + 3y + z &= 1 \\ z - y &= 0 \\ 2y + 3x - 2 &= z \end{aligned}$$

Exercise 40: Write a function in an m-file of name `gauss_pivot.m` which implements Gauss's reduction algorithm with partial pivoting (i.e. pivoting in rows). The output should include at least the upper triangular matrix and the transformed vector of independent terms. Bonus points for returning also the lower triangular and the permutation matrices. Recall that this algorithm is also called *LUP* factorization. Use this function to compute the reduced form of the systems in Exercise 39. —

Exercise 41: Using either the code from Listing 6.1 or the one you produced for Exercise 40, write a new function which explicitly *solves* a linear system of equations using Gauss's method. This requires computing the values of each variable *after* the reduction step and returning them as a vector. Call that function `gauss_solve`, write it in an m-file and use it for solving the systems in Exercise 39. —

2. Iterative algorithms

Iterative algorithms are much simpler to implement, as they only need an iteration of a simple matrix multiplication. They are also much faster than exact methods for large systems and can produce high-quality approximations to the solutions. As a matter of fact, the name *exact methods* for Gauss-type algorithms is a misnomer, as rounding errors are always present and floating-point operations are almost never

exact. This makes iterative algorithms as powerful as those methods and, in real life, much more useful.

Even though they require the computation of the inverse of a matrix, this does not complicate the methods too much because the matrix to be inverted is “simple” (either a *diagonal* one, in Jacobi’s or a triangular one for Gauss-Seidel). In what follows, we shall make use of the inversion utility of Matlab (although, to be honest, we should implement inversion ourselves, but this would complicate matters too much to little avail).

2.1. Jacobi’s Method. Given a linear system of equations

$$(4) \quad Ax = b$$

Jacobi’s method consists in transforming it into a fixed-point problem by rewriting A as the sum of a diagonal matrix and another one. Let D be the diagonal matrix whose diagonal coincides with that of A . Then $A = D + N$, where N has only 0 on the diagonal. System (4) is then

$$(D + N)x = b$$

If D is invertible, we can rewrite this as

$$D^{-1}(D + N)x = D^{-1}b,$$

which, expanding the left hand side gives

$$x + D^{-1}Nx = D^{-1}b$$

and finally, moving the second term of the addition to the right hand side, we get

$$x = D^{-1}b - D^{-1}Nx.$$

This means that, if we call f to the transformation

$$f(x) = D^{-1}b - D^{-1}Nx,$$

system (4) is equivalent to the fixed-point problem

$$f(x) = x.$$

It is known that, under certain conditions, given *any* initial seed x_0 , the iteration $x_1 = f(x_0)$, $x_i = f(x_{i-1})$ converges to a solution of the problem. This means that the following steps are a correct description of Jacobi’s method.

- (1) Set x_0 to any vector.
- (2) Let $k = 1$, N be a bound for the number of steps and ϵ a tolerance.
- (3) Let $x_1 = D^{-1}b - D^{-1}Nx_0$
- (4) While $k < N$ and $\|x_k - x_{k-1}\| > \epsilon$ do:

- $k = k + 1$
 - $x_k = D^{-1}b - D^{-1}Nx_{k-1}$.
- (5) If $k == N$ finish with an error (tolerance not reached), otherwise return x_k .

Exercise 42: Implement Jacobi's method using the rough description above. Call the file `jacobi.m` and use it to solve the systems of Exercise 39. Compare the approximate solutions obtained for different tolerances with those of Exercise 41.

Remark: Notice that both N and ϵ are required as input to the function `jacobi` and that default values should be given for them. —

Exercise 43: Use the `rand` function to create a 100×100 matrix A and a 100×1 vector b . Use the `gauss` and `jacobi` functions of exercises 41 and 42 to compute two solutions, x_1 and x_2 . Compare them between themselves and compare them to any solution found using Matlab's solver. Which do you think is better? —

2.2. The Gauss-Seidel Method. The Gauss-Seidel method is very similar to Jacobi's. Instead of transforming System (4) using the diagonal of A , one takes its lower triangular part and writes

$$(L + M)x = b$$

where L is lower triangular and M is upper triangular with 0 on its diagonal. If L is invertible, this equation can be rewritten

$$L^{-1}(L + M)x = L^{-1}b,$$

which, after transformations similar to those made for Jacobi's method, turns into

$$x = L^{-1}b - L^{-1}Mx,$$

which is a fixed-point problem.

Exercise 44: Implement the Gauss-Seidel method using a file called `gauss_seidel.m`. Use it to solve the systems of Exercise 39 and compare your solutions with those of exercises 41 and 42. —

Exercise 45: Use the `random` utility of Matlab to create a 100×100 matrix A and a 100×1 vector b . Use the `gauss_seidel` function of Exercise 44 to compute a solution x_1 and the `jacobi` function of Exercise 42 to compute another one x_2 . Compare them. Which is better? —

CHAPTER 7

Approximate solutions to nonlinear equations

This chapter deals with finding approximate solutions to equations of one variable:

$$(5) \quad f(x) = 0$$

where f is a “reasonably behaved” function. The meaning of this expression depends on the context but one usually requires f to be *at the very least*, continuous.

1. Bisection method

One of the first methods for finding approximate roots of *continuous* functions is based on Bolzano’s theorem. Given a continuous function $f : [a, b] \rightarrow \mathcal{R}$ with $f(a)f(b) < 0$ (that is, which changes sign between a and b), there must be a point $c \in (a, b)$ such that $f(c) = 0$. From this, one infers that either $z = (a + b)/2$ is a root of f or f satisfies the same conditions on either the left interval $[a, z]$ or the right one $[z, b]$. This lets one repeat the procedure (taking the midpoint) until a tolerance is reached. This method is called the Bisection algorithm. It can be described more formally as in Algorithm 5.

Exercise 46: Implement the Bisection algorithm in an m-file called `bisection.m`. —

Exercise 47: Use the bisection algorithm to find approximate roots to the following functions in the specified intervals. Notice that the algorithm may not work for some of them even though they have roots in that interval.

$$\begin{aligned} f(x) &= \cos(e^x), x \in [0, 2] \\ g(x) &= x^3 - 2, x \in [0, 3] \\ h(x) &= e^x - 2 \cos(x), x \in [0, \pi] \\ r(t) &= t^2 - 1, x \in [-2, 2] \\ s(z) &= \sin(z) + \cos(z), z \in [0, 2\pi] \\ u(t) &= t^2 - 3, t \in [-2, 2] \\ f(t) &= \operatorname{atan}(t - 2), t \in [-3, 3] \end{aligned}$$

Algorithm 5 Bisection Algorithm.

Input: A function $f(x)$, a pair of real numbers a, b with $f(a)f(b) < 0$, a tolerance $\epsilon > 0$ and a limit of iterations $N > 0$

Output: either an error message or a real number c between a and b such that $|f(c)| < \epsilon$ (i.e. an approximate root)

★PRECONDITION

if $f(a) \notin \mathbb{R}$ **of** $f(b) \notin \mathbb{R}$ **then**

return ERROR

end if

★START

$i \leftarrow 0$

$c \leftarrow \frac{a+b}{2}$

while $|f(c)| \geq \epsilon$ **and** $i \leq N$ **do**

if $f(a)f(c) < 0$ **then**

$b \leftarrow c$ [interval $[a, c]$]

else

$a \leftarrow c$ [interval $[c, b]$]

end if

$i \leftarrow i + 1$

$c \leftarrow \frac{a+b}{2}$ [middle point]

end while

if $i > N$ **then**

return ERROR

end if

return c

Can you use a different interval for those functions for which the algorithm does not work in order to find a root of them? —

2. The Newton-Raphson method

The Newton-Raphson method for finding roots of an equation $f(x) = 0$ requires f to be, at least, differentiable (and hence, continuous). It can be described as in Algorithm 6.

Example 18 In order to implement the Newton-Raphson method *without using symbolic operations* in Matlab, one needs to define a function whose input includes the seed $\mathbf{x0}$, the function \mathbf{f} (as an anonymous function), its derivative \mathbf{fp} (because one is not using symbolic derivation) and both the tolerance `epsilon` and the limit for the iterations N . Listing 7.1 shows a possible implementation.

Algorithm 6 Newton-Raphson.

Input: A differentiable function $f(x)$, a *seed* $x_0 \in \mathbb{R}$, a tolerance $\epsilon > 0$ and a limit for the number of iterations $N > 0$

Output: Either an error message or a real number c such that $|f(c)| < \epsilon$ (i.e. an approximate root)

```

*START
i ← 0
while |f(xi)| ≥ ε and i ≤ N do
    xi+1 ← xi -  $\frac{f(x_i)}{f'(x_i)}$  [possible NaN or ∞]
    i ← i + 1
end while
if i > N then
    return ERROR
end if
c ← xi
return c

```

```


% Newton-Raphson implementation.
% Returns both the approximate root and the number of
% iterations performed.
% Input:
% f, fp (derivative), x0, epsilon, N
function [z n] = newtonraphson(f, fp, x0, epsilon, N)
    n = 0;
    xn = x0;
    % initialize z to a NaN (i.e. error by default)
    z = NaN;
    % Both f and fp are anonymous functions
    fn = f(xn);
    while(abs(fn) >= epsilon && n <= N)
        n = n + 1;
        fn = f(xn); % memorize to prevent recomputing
        % next iteration
        xn = xn - fn/fp(xn); % an exception might take place here
    end
    z = xn;
    if(n == N)
        warning('Tolerance not reached.');
```


LISTING 7.1. A possible (simple) implementation of the Newton-Raphson method.

Notice that, as the function is called `newtonraphson`, the file should be called `newtonraphson.m`. 

Exercise 48: Use the `newtonraphson` function defined in Example 18 to compute approximate solutions to the functions of Exercise 47. Use different values for the seed, epsilon and limit of iterations.

Does the Newton-Raphson method converge always? 

Exercise 49: Modify the code of `newtonraphson` (and create a new function called `newtonraphson_plot`, and the corresponding file) so that, at each step, the point $(x_i, 0)$ is plotted on a graph, with a big enough mark. Use this to keep track of different calls to the function for the equations of Exercise 47. 

Exercise 50: Remark: only do this exercise if you have enough time to spare. As in Exercise 49, create a new function called `newtonraphson_graph` which plots not only each $(x_i, 0)$ but also the tangent line from $(x_{i-1}, f(x_{i-1}))$ to $(x_i, 0)$ to visualize the process. Use the command `pause` between each plot to wait for the user to press some key (so that the process is seen step by step). Use this function for the equations in Exercise 47. 


3. The Secant method

The secant method is a slight modification of the Newton-Raphson algorithm when the derivative of f is either unknown or too expensive to compute. Instead of using $f'(x_i)$ at each step, the algorithm approximates that value as

$$f'(x_i) \simeq \frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}},$$

which requires keeping track not only of x_i but also of x_{i-1} at each step.

Exercise 51: Write a function `secant` in a file called `secant.m` which implements the secant algorithm. Notice that, unlike the Newton-Raphson function, this requires only an anonymous function `f` as input but two seeds `x0` and `x1` (for example).

Use this `secant` function to compute approximate roots to the equations of Exercise 47. 

4. The fixed-point method

The fixed-point method is based on the following result

THEOREM 1. *Let $g : [a, b] \rightarrow [a, b]$ be a map of $[a, b]$ into itself, which is continuous and differentiable on $[a, b]$. If there is a positive $\lambda < 1$ such that for any $x \in [a, b]$, $|g'(x)| \leq \lambda$, then there exists one and only one $\alpha \in [a, b]$ for which $g(\alpha) = \alpha$.*

That requires verifying quite a few conditions before it can be used for solving an equation (and, as a matter of fact, turning that equation into a fixed-point problem). Essentially, if one starts with

$$f(x) = 0,$$

in order to use the fixed-point method one has to transform that equation into a “fixed-point problem,” that is, an equation of the form

$$\tilde{f}(x) = x$$

(which means that one is looking for a value c which is *fixed* by \tilde{f} : $\tilde{f}(c) = c$, whence the name). The usual transformation is easy

$$f(x) = 0 \Leftrightarrow f(x) + x = x.$$

Obviously $f(c) = 0$ if and only if $f(c) + c = c$. Thus, generally, $\tilde{f}(x) = f(x) + x$.

Instead of trying to verify all the necessary conditions in Theorem 1, one may simply try to find a solution to $f(x) + x = x$, given an equation $f(x) = 0$.

Exercise 52: Write a function `fixed_point` which implements the fixed-point method. The input should be an anonymous function `f`, a seed `x0`, a tolerance `epsilon` and a limit for the number of iterations `N`. The output should be either a warning or error or a number `c` such that `abs(f(c)) < epsilon`.

Remark: the iteration for this problem is not just $x_{i+1} = f(x_i)$ because we are looking for a *root* of $f(x)$!

Use this function to compute (if possible) approximate roots to the equations of Exercise 47. Does it work? When? Why?

APPENDIX A

Program structure

We describe the *basic* structure of the function files (.m files implementing Matlab functions) which appear in this course. This is a *very* elementary type of file but serves as a simplification of the implementation of an algorithm in a .m file.

We assume the function is called `fName` and it has the following:

- Input Parameters: $p_1, p_2, p_3, \dots, p_n$.
- Output Values: o_1, o_2, \dots, o_n .

For instance, the Bisection Algorithm requires 5 input parameters:

- f**: the function of which to find a root,
- a**: the left endpoint of the interval,
- b**: the right endpoint,
- N**: the limit to the number of iterations,
- e**: the tolerance.

and it may be implemented to give 2 output values:

- r**: the approximate root,
- k**: the number of iterations performed.

A program with the input and output described above will have a structure similar to the code in Listing A.1 and *must be saved in a file called fName.m*.

```
function [o1, o2, ..., on] = fName(p1, p2, ..., pn)
    o1 = reasonable default value
    o2 = reasonable default value
    % ...
    % and so on until all the output values have been initialized

    % if a counter is needed, initialize it here
    k = 1

    % Some computations may have to be performed before the main
    % loop, they would go here. For example: the FIRST iteration
    % may have to be carried out BEFORE starting the loop.

    % Now a WHILE or FOR loop will come.
    % It will generally be a WHILE() unless the number of iterations
```

```

% or the set of values on which to iterate is already known. Or
% In a WHILE loop there will be several stopping conditions, the
% last one will be of the form
%     k < Number of Iterations
% IF THERE IS A COUNTER, obviously

while( CONDITION1 && CONDITION2 ... && CONDITIONn )

    % The body of the algorithm goes here, this is
    % "The algorithm itself".
    % (bisection, Newton-Raphson, Euler...)
    %
    % Here one may find ifs, assignments, anything

end

% Before finishing, one has to verify that the answer is
% at least coherent:
% If there is a counter and the maximum number of iterations
% has been reached, then one MUST warn the user of this fact
% like this:
if( k >= N) % assuming the maximum number is N
    warning('Maximum number of iterations reached')
end

% end of the program, do not forget the trailing 'end'
end

```

LISTING A.1. Basic structure of an elementary program.