Lecture Notes on Numerical Methods for Engineering (?)

Pedro Fortuny Ayuso

Universidad de Oviedo
E-mail address: `fortunypedro@uniovi.es`

Contents

Introduction

These notes cover what is taught in the classes of Numerical Methods for Engineering in the School at Mieres. One should not expect more than that: a revision of what has been or will be explained during the course. For a more correct, deep and thorough explanation, one should go to the material referenced in the bibliography*

There is none.

Simply, stated

These notes are no substitute for a book (or two).

and even more

What is here may and may not cover completely the contents of the exam

1. Some minor comments

My aim in these notes is mostly twofold:

- To introduce the basic problems tackled by Numerical Calculus in their most simple fashion.
- To get the students used to stating algorithms with precision and to understanding the idea of complexity.

I also would like to be able to make the students aware of the importance of the *conditioning* of a numerical problem and the need to verify that the results obtained by a computer program *match* reality and fit the problem under study. But there is really no space (much less time in the course) for this.

We shall refer, along the way, to several computing tools. Specifically,

Matlab: I shall try to make all the code in this notes runnable on Octave but this text will only speak of Matlab, which is the software students are used to working with at Mieres.

Wolfram Alpha: This is a powerful and very useful tool with which a large number of operations (mathematical and not) can be performed, on a simple web page.

What may be most interesting is that the students can:

- State algorithms *with precision*, aware that an algorithm must be *clear*, *finite* and *finish*.
- Follow the steps of an algorithm and roughly analyze its complexity (or at least, compare two of them).

I would also like to stress the importance of discerning if an approach suits a specific problem or not depending on its good or bad conditioning but, alas, there is also too little time in the course…

It goes without saying that they will have to understand and be able to follow, step by step, the algorithms stated in the classes (this is an essential requisite). All the algorithms that will be explained are of easy memorization because they are either brief (most of them) or geometrically elemental (so that their verbal expression is easily to remember). Formal precision is much more important in this course than geometric ideas because numerical analysis deals with *formal* methods of solving specific problems, not with their geometrical or intuitive expression. This is the main *rub* of this course. You will need to *memorize*.

CHAPTER 1

Arithmetic and error analysis

Exponential notations, double precision floating point, the notion of error and some common sources of error in computations are summarily reviewed. The student should at least get a feeling of the importance of *normalization* of floating point arithmetic and that mistakes in its use can be critical.

1. Exponential notation

Real numbers can have a finite or infinite number of digits. When working with them they have to be approximated using a finite (and usually small) number of digits. Even more, they should be expressed in a standard way so that their magnitude and significant digits are easily noticed at first sight and so that sharing them among machines is a deterministic and efficient process. These requirements have led to what is known as *scientific* or *exponential* notation. We shall explain it roughly; what we intend is to transmit the underlying idea more than to detail all the rules (they have lots of particularities and exceptions). We shall use, along these notes, the following expressions:

DEFINITION 1. An *exponential notation* of a real number is an expression of the form

$$\pm A.B \times 10^C$$
$$10^C \times \pm A.B$$
$$\pm A.B\mathbf{e}C$$

where A, B and C are natural numbers (possibly 0), $\pm$ is a sign (which may be elided if $+$). Any of those expressions refers to the real number $(A + 0.B) \cdot 10^C$ (where $0.B$ is "nought dot B…").

For example:

- The number 3.123 is the same 3.123.
- The number $0.01e-7$ is 0.000000001 (eight zeroes after the dot and before the 1).

- The number $10^3 \times -2.3$ is -2300.
- The number $-23.783e-1$ is -2.3783.

In general, scientific notation assumes the number to the left of the decimal point is a single non-zero digit:

DEFINITION 2. The *standard exponential notation* is the exponential notation in which A is between 1 and 9.

Finally, machines usually store real numbers in a very specific way called *floating point*.

DEFINITION 3. A *floating point format* is a specification of an exponential notation in which the length of A plus the length of B is constant and in which the exponents (the number C) vary in a specific range.

Thus, a floating point format can only express a finite amount of numbers (those which can be written following the specifications of the format).

The blueprint for the floating point standard for microchips (and for electronic devices in general) is document IEEE-754 (read "I-E-cube seven-five-four"). The acronym stands for "Institute of Electrical and Electronic Engineers". The last version of the document dates from 2008.

1.1. The binary double precision format of IEEE-754.

The double precision format is the IEEE specification for representing real numbers in sequences of 16, 32, 64, 128 bits (and more cases) and their decimal representation. The main properties of binary double precision with 64 bits are roughly explained in this section.

In order to write numbers in double precision, 64 bits are used, that is, 64 *binary digits*. The first one stands for the sign (a 0 means $+$, a 1 means $-$). The 11 following ones are used for the exponent (as will be shown) and the remaining 52 are used for what is called the *mantissa*. Thus, a double precision number has three parts: s, the sign, which can be 0 or 1; e, the exponent, which varies between 0 and $2^{11}-1=2047$; and m, a 52-bit number. Given three values for s, e and m, the real number represented by them is:

- If $e \neq 0$ and $e \neq 2047$ (i.e. if the exponent is not one of the end values), then
$$N = (-1)^s \times 2^{e-1023} \times 1.m,$$
where $1.m$ means "one-dot-m" in binary. Notice —and this is the key— that the exponent *is not the number represented by*

the 11 *bits of e* but that it is "shifted to the right". The exponent $e = 01010101011$, which in decimal is 683 represents, in double precision format, the number $2^{683-1023} = 2^{-340}$. Those e with a starting 0 bit correspond to negative powers of 2 and those having a starting 1 bit to positive powers (recall that $2^{10} = 1024$).

- If $e = 0$ then, if $m \neq 0$ (if there is a mantissa):
$$N = (-1)^s \times 2^{-1023} \times 0.m,$$
where $0.m$ means "zero-dot-m" in binary.
- If $e = 0$ and $m = 0$, the number is either $+0$ or -0, depending on the sign s. This means that 0 has a sign in double precision arithmetic (which makes sense for technical reasons).
- The case $e = 2047$ (when all the eleven digits of e are 1) is reserved for encoding "exceptions" like $\pm\infty$ and NaN (not-a-number-s). We shall not enter into details.

As a matter of fact, the standard is much longer and thorough and includes a long list of requisites for electronic devices working in floating point (for example, it specifies how truncations of the main mathematical operations have to be performed to ensure exactness of the results whenever possible).

The main advantage of floating point (and of double precision specifically) is, apart from standardization, that it enables computing with very small and very large numbers with a single format (the smallest storable number is $2^{-1023} \simeq 10^{-300}$ and the largest one is $2^{1023} \simeq 10^{300}$). The trade off is that if one works *simultaneously* with both types of quantities, the first lose precision and tend to disappear (a *truncation error* takes place). However, used *carefully*, it is a hugely useful and versatile format.

1.2. Binary to decimal (and back) conversion. * In this course, it is *essential* to be able to convert a number from binary to decimal representation and back.

Examples?

In these notes, we shall use the following notation: the expression $x \leftarrow a$ means that x is a variable, a is a value (so that it can be a number or another variable) and that x gets assigned the value of a. The expression $u = c$ is the *conditional* statement "the value designed by u is the same as that designed by c", which can be either true or false. We also denote $m//n$ as the quotient of dividing the natural number $m > 0$ by the natural number $n > 0$, and $m\%n$ is the *remainder* of that division. That is,

$$m = (m//n) \times n + (m\%n).$$

Finally, if x is a real number $x = A.B$, the expression $\{x\}$ means the *fractional part of* x, i.e., the number $0.B$.

EXAMPLE 1. The following table shows the decimal and binary expansions of several numbers. The last binary expansion is approximate because it has an infinite number of figures.

Decimal	*Binary*
1	1
2	10
10	1010
0.5	0.1
7.25	111.01
0.1	0.000110011+

Algorithm 1 (in page 11) is a way to convert a decimal number $A.B$ with a finite number of digits to its binary form. Algorithm 2 performs the reverse conversion. Notice that, as there are numbers with a finite quantity of decimal digits which cannot be expressed with a finite quantity of binary digits (the most obvious example being 0.1), one has to specify a number of fractional digits for the output (which implies that one does not necessarily obtain the very same number but a truncation).

Converting from binary to decimal is "simpler" but requires additions on the go: one does not obtain a digit at each step because one has to add powers of two. This process is described in Algorithm 2. Notice that the number of digits of the output is not specified (it could be done but it would only make the algorithm more complex). Finally, in all those steps in which $A_i \times 2^i$ or $B_i \times 2^{-i}$ is added, both A_i and B_i are either 0 or 1, so that this multiplication just means "either add or do not add" the corresponding power of 2 (this is what a binary digit means, anyway).

2. Error, basic definitions

Whenever numbers with a finite quantity of digits are used in computations and whenever real measurements are performed, one has to acknowledge that errors will be made. This is not grave by itself. What matters is *having an estimate of their size* and knowing that, as calculations are made, they can propagate. In the end, the best one can do is *to bound the absolute error*, that is, to know a *reasonable*

Algorithm 1 Conversion from decimal to binary, without sign.

```
  **Input:** $A.B$ a number in base 10, with $B$ of finite length, $k$ a positive integer (which is the desired number of fractional digits after the binary dot)
  **Output:** $a.b$ (the number $x$ in binary format, truncated to $2^{-k}$)
  **if** $A = 0$ **then**
      $a \leftarrow 0$ and go to the FRACTIONAL PART
  **end if**
⋆INTEGRAL PART
  $i \leftarrow -1, n \leftarrow A$
  **while** $n > 0$ **do**
      $i \leftarrow i + 1$
      $x_i \leftarrow n\%2$ [remainder]
      $n \leftarrow n//2$ [quotient]
  **end while**
  $a \leftarrow x_i x_{i-1} \dots x_0$ [the sequence of remainders in reverse order]
⋆FRACTIONAL PART
  **if** $B = 0$ **then**
      $b \leftarrow 0$
      **return** $a.b$
  **end if**
  $i \leftarrow 0, n \leftarrow 0.B$
  **while** $n > 0$ **and** $i < k$ **do**
      $i \leftarrow i + 1$
      $m \leftarrow 2n$
      **if** $m \geq 1$ **then**
          $b_i \leftarrow 1$
      **else**
          $b_i \leftarrow 0$
      **end if**
      $n \leftarrow \{m\}$ [the fractional part of $m$]
  **end while**
  $b \leftarrow b_1 b_2 \dots b_i$
  **return** a.b
```

value (the bound) which is larger than the error, in order to assess, with certainty, how far the real value can be from the computed one.

In what follows, an exact value x is assumed (a constant, a datum, the exact solution to a problem…) and an approximation will be denoted $\tilde{x}$.

Algorithm 2 Conversion from binary to decimal, without sign.

```
  Input: $A.B$, a number in binary format, $k$ a non-negative integer
  (the number of the fractional binary digits to be used).
  Output: $a.b$, the decimal expression of the truncation up to preci-
  sion $2^{-k}$ of the number $A.B$
⋆INTEGRAL PART
  Write $A = A_r A_{r-1} \dots A_0$ (the binary digits)
  $a \leftarrow 0, i \leftarrow 0$
  while $i \leq r$ do
      $a \leftarrow a + A_i \times 2^i$
      $i \leftarrow i + 1$
  end while
⋆FRACTIONAL PART
  if $B = 0$ then
      return $a.0$
  end if
  $b \leftarrow 0, i \leftarrow 0$
  while $i \leq k$ do
      $i \leftarrow i + 1$
      $b \leftarrow b + B_i \times 2^{-i}$
  end while
  return $a.b$
```

DEFINITION 4. The *absolute error* incurred when using $\tilde{x}$ instead of x is the absolute value of their difference $|x - \tilde{x}|$.

But, unless x is 0, one is usually more interested in the *order of magnitude* of the error; that is, "how *relatively* large the error is" as compared to the real quantity.

DEFINITION 5. The *relative error* incurred when using $\tilde{x}$ instead of x, assuming $x \neq 0$, is the quotient

$$\frac{|\tilde{x} - x|}{|x|}$$

(which is always positive).

We are not going to use a specific notation for them (some people use Δ and δ, respectively).

EXAMPLE 2. The constant π, which is the ratio between the length of the circumference and its diameter, is, approximately $3.1415926534+$ (the trailing $+$ indicates that the real number is larger than the representation). Assume one uses the approximation $\tilde{\pi} = 3.14$. Then

- The absolute error is $|\pi - \tilde{\pi}| = 0.0015926534+$.
- The relative error is $\frac{|\pi-\tilde{\pi}|}{\pi} \simeq 10^{-4} \times 5.069573$.

This last statement means that one is incurring an error of 5 ten-thousandths per unit (approx. 1/2000) each time 3.14 is used for π. Thus, if one adds 3.14 two thousand times, the error incurred when using this quantity instead of $2000 \times \pi$ will be approximately π. This is the meaning of the relative error: its reciprocal is the number of times one has to add $\tilde{x}$ in order to get an accrued error equal to the number being approximated.

Before proceeding with the analysis of errors, let us define the two most common ways of approximating numbers using a finite quantity of digits: *truncation* and *rounding*. The precise definitions are too detailed for us to give. Start with a real number (with possibly an infinite quantity of digits):

$$x = a_1 a_2 \dots a_r \,.\, a_{r+1} \dots a_n \dots$$

notice that there are r digits to the left of the decimal point. Define:

DEFINITION 6. The *truncation of x* to k (significant) digits is:

- the number $a_1 a_2 \dots a_k 0 \dots 0$ (an integer with r digits), if $k \leq r$,
- the number $a_1 \dots a_r.a_{r+1} \dots a_k$ if $k > r$.

That is, one just cuts (i.e. *truncates*) the numerical expression of x and adds zeroes to the right if the decimal part has not been reached.

DEFINITION 7. The *rounding of x* to k (significant) digits is the following number:

- If $a_{k+1} < 5$, then the rounding is the same as the truncation.
- If $5 \leq a_{k+1} \leq 9$, then the rounding is the truncation plus 10^{r-k+1}.

Remark: the rounding described here is called *biased to plus infinity* because when the $k+1-$th digit is greater than or equal to 5, the approximation is always greater than the true value.

The problem with rounding is that *all the digits of the rounded number* can be different from the original value (think of 0.9999 rounded to 3 significant digits). The great advantage is that the error incurred when rounding is less than the one incurred when truncating (it can even be a half of it):

EXAMPLE 3. If $x = 178.299$ and one needs 4 significant[1] figures, then the truncation is $x_1 = 178.2$, whereas the rounding is 178.3. The absolute error in the former case is 0.099, while it is 0.001 in the latter.

EXAMPLE 4. If $x = 999.995$ and 5 digits are being used, the truncation is $x_1 = 999.99$, while the rounding is 1000.0. Even though all the digits are different the absolute error incurred in both cases is the same: 0.005. This is what matters, not that the digits "match".

Why is then truncation important if rounding is always at least as precise and half the time better? Because when using floating point, one cannot prevent truncation from happening and it must be taken into account to bound the global error. As of today (2013) most of the computer programs working in double precision, perform their computations internally with many more digits and round their results to double precision. However, truncations always happen (there is a limited amount of available digits to the processor).

2.1. Sources of error. Error appears in different ways. On one hand, any measurement is subject to it (this why any measuring device is sold with an estimated margin of precision); this is intrinsic to Nature and one can only take it into account and try to assess its magnitude (give a bound). On the other hand, computations performed in finite precision arithmetic both propagate these errors and give rise to new ones precisely because the quantity of available digits is finite.

The following are some of the sources of error:

- *Measurement error*, already mentioned. This is unavoidable.
- *Truncation error*: happens whenever a number (datum or the result of a computation) has more digits than available and some of them must be "forgotten".
- *Rounding error*: takes place when a number is rounded to a specified precision.
- *Loss of significance* (also called *cancellation error*): this appears when an operation on two numbers increases relative error substantially more than the absolute error. For example, when numbers of very similar magnitude are subtracted. The digits lost to truncation or rounding are too relevant and the relative error is huge. A classical example is the *instability of the solution of the quadratic equation*.

[1]We have not explained the meaning of this term, but we shall not do so. Suffice it to say that "significant figures" means "exact figures" in an expression.

- *Accumulation error*: which appears when accumulating (with additions, essentially) small errors of the same sign *a lot of times*. This is what happened with the Patriot Missile Stations in 1991, during the *Desert Storm* operation[2].

All the above errors may take place when working with finite arithmetic. The following rules (which describe the worst-case behaviors) apply:

- When adding *numbers of the same sign*, the absolute error can be up to the sum of the absolute errors and the same can happen to the relative error.
- When adding *numbers of different sign*, the absolute error behaves like in the previous case but *the relative error may behave wildly*: $1000.2 - 1000.1$ has only a significant digit, so that the relative error can be up to 10%, but the relative error in the operands was at most 1×10^{-4}.
- When multiplying, the absolute error is of the same magnitude as the largest factor times the absolute error in the other factor. If both factors are of the same magnitude, the absolute error can be up to double the maximum absolute error times the maximum factor. The relative error is of the same magnitude as the maximum relative error in the factors (and is the sum if both factors are of the same magnitude).
- When dividing *by a number greater than or equal to* 1, the absolute error is approximately the absolute error in the numerator divided by the denominator and the relative error is the relative error in the numerator (more or less like in multiplication). When dividing by numbers near 0, absolute precision is lost and later operations with numbers of the same magnitude will probably give rise to large cancellation errors. Example 5 is illustrative of this fact: the first line is assumed to be an exact computation, the second one an approximation using a truncation to 6 significant figures in the divisor of the quotient. One can see that the "approximation" is totally useless.

An example of the bad behavior of division and addition is the following:

[2]One can read a summary at
`http://www.cs.utexas.edu/~downing/papers/PatriotA1992.pdf`
and the official report is also available:
`http://www.fas.org/spp/starwars/gao/im92026.htm`.

EXAMPLE 5. Consider the following two computations, the first one is assumed to be "correct" while the second one is an approximation:

$$26493 - \frac{33}{0.0012456} = -0.256 \text{ (the exact result)} .$$
$$26493 - \frac{33}{0.001246} = 8.2488 \text{ (approximation)}$$

The relative error in the rounding of the denominator is only 3×10^{-4} (less than half a thousandth of a unit), but the relative error in the final result is 33.2 (which means that the obtained result is 33 times larger in absolute value than it should, so it is worthless as an "approximation"). Notice that *not even the sign is correct*. This is (essentially) the main problem of resolution methods involving divisions (like Gauss' and, obviously, Cramer's). When explaining Gauss' method, convenient ways to choose an adequate denominator will be shown (these are called *pivoting strategies*). As a general rule, *the larger the denominator, the better*.

3. Bounding the error

As we have already said, one does not seek an exact knowledge of the error incurred during a measurement or the solution to a problem, as this will likely be impossible. Rather, one aims to *have an idea* of it and, especially, to *know a good bound* of it. That is, to be able to assess the maximum value that the absolute error can take and this being a useful piece of information. Knowing that 2.71 is an approximation of e with an error less than 400 is worthless. Knowing that the error is less than 0.01 is useful.

In general, the only possibility is to estimate a *reasonably small* number larger than the incurred error. This is *bounding* the error.

3.1. Some bounds. If one knows that a quantity is between two values, which is usually expressed in the form $x = a \pm \epsilon$ for some $\epsilon > 0$, then the absolute error incurred when using a instead of x is unknown *but is bounded by* ϵ. So that the relative error is, at most, this ϵ divided by the smallest of the possible absolute values of x. *Notice that this is tricky because if* $a - \epsilon < 0$ *and* $a + \epsilon > 0$ *then* there is no way to bound the relative error *because* x *can be* 0 *and then there is no relative error (it is not defined for* $x = 0$*).*

EXAMPLE 6. If $\pi = 3.14 \pm 0.01$, then the maximum absolute error incurred is 0.01, so that the relative error is at most

$$\frac{0.01}{|3.13|} \simeq .003194$$

(more or less 1/300).

Notice that in order to get an upper bound of a quotient, *the denominator must be bounded from below* (the lesser the denominator, the greater the quotient).

The rules in page 15 are essential in order to bound the error of a sequence of arithmetic operations. One has to be especially careful when dividing by numbers less than one, as this may lead to useless results like “the result is 7 with a relative error of 23.”

CHAPTER 2

Numerical Solutions to Non-linear Equations

The problem of solving the non-linear equation $f(x) = 0$ given f and some initial conditions is treated in this chapter.

Each of the algorithms which will be explained in this chapter has its own advantages and disadvantages; one should not discard anyone *a priori* just for its "slowness" —for example, bisection. We shall see that the "best" one —namely, Newton-Raphson's— has two drawbacks: it may converge far from the initial condition, becoming useless for finding a root "near" a specific point and it requires the computation of the value of the derivative of f at each step, which may be too costly.

Notice that all the algorithms we present include stopping conditions which depend *on the value of $f(x)$* and are of Cauchy-type. This is done in order to simplify the exposition. If a specific quantity of exact figures is required, one needs to perform a much deeper study of the problem being solved, of the derivative of f and other elements, which are out of the scope of these notes.

1. Introduction

Computing roots of functions, and especially of polynomials, is one of the classical problems of Mathematics. It used to be believed that any polynomial could be "explicitly solved" like the quadratic equation, via a formula involving radicals (roots of numbers). Galois Theory, developed in the beginning of the XIX Century, proved that this is not the case at all and that, as a matter of fact, most polynomials of degree greater than 4 are not solvable using radicals.

However, the search for a *closed formula* for solving equations is just a way of putting off the real problem. In the end, and this is what matters:

The only computations that can always be performed exactly are addition, subtraction and multiplication.

It is not even possible to divide and get exact results[1].

From the beginning, people sought ways to quickly find approximate solutions to nonlinear equations involving the four rules: addition, subtraction, multiplication and division.

Two different kind of algorithms are explained in this chapter: those with a geometric meaning and the *fixed point* method, which requires the notion of *contractivity*.

Before proceeding, let us remark two essential requirements in any root-computing algorithm:

- The specification of a *precision*, that is an $\epsilon > 0$ such that if $|f(c)| < \epsilon$, then c is taken as an approximate root of f. This is needed because, as non-exact computations are performed, expecting a result $f(c) = 0$ is unrealistic, so that this equality is useless as a stopping condition.
- A second *stopping condition* unrelated to the value of f. It may well happen that either the algorithm does not converge or that f has no roots, so that the statement $|f(c)| < \epsilon$ is never true. In any case, the algorithm *must* finish at some point. Otherwise, *it is not an algorithm*. This implies the need for a condition unrelated to f. This takes usually the form "perform no more than n steps," where n is a counter.

2. The Bisection Algorithm

Assume f is a continuous function on a closed interval $[a, b]$. Bolzano's Theorem may be useful if its conditions hold:

THEOREM (Bolzano). *Let $f : [a, b] \to \mathbb{R}$ be a continuous function on $[a, b]$ such that $f(a)f(b) < 0$ (i.e. it changes sign between a and b). Then there is $c \in (a, b)$ with $f(c) = 0$.*

This statement asserts that if the sign of f changes on $[a, b]$ then there is *at least* a root of f in it. One could sample the interval using small sub-intervals (say of width $(b - a)/10^i$) and seek, among this sub-intervals, one where the sign changes, thus nearing a root at each step. Actually, dividing the interval into two segments turns out to be much simpler.

Start with a function f, *continuous* on the interval $[a, b]$ with values $f(a)$ and $f(b)$. Specify a desired precision $\epsilon > 0$ (so that if

[1]One could argue that using rational numbers solves this problem but, again, there is a point at which decimal expansions are needed.

$|f(c)| < \epsilon$ then c is accepted as an approximate root) and a maximum number of iterations $N > 0$ of the process. With all these data, one proceeds taking advantage of Bolzano's Theorem, as follows:

If $f(a)f(b) < 0$, then there is a root in (a, b). Take c as the midpoint[2] of the interval, that is $c = \dfrac{a+b}{2}$. There are three possibilities now:

- Either $|f(c)| < \epsilon$ and one takes c as an approximate root and finishes the algorithm.
- Or $f(a)f(c) < 0$ and one substitutes c for b and repeats.
- Or $f(c)f(b) < 0$ and one substitutes c for a and repeats.

The iteration is done at most N times (so that one obviously has to keep track of their number). If after N iterations no approximate root has been found, the process ends with an error message.

The formal statement of the process just described is Algorithm 3. Notice that the expression $a \leftarrow b$ is used, meaning that a takes (or is given) the value b.

EXAMPLE 7. Take the function $f(x) = \cos(e^x)$, which is continuous in the interval $[0, 1]$. Obviously, $f(0)f(1) < 0$ (why is this obvious?), so that Bolzano's Theorem applies. Setting $a = 0$, $b = 1$, Algorithm 3 gives the following sequence: each assignment to `a` or `b` means that the corresponding endpoint of the interval is substituted for that value.

```
octave > Bisec(@(x) cos(exp(x)), 0, 1, .001, 10)
c = 0.50000
    b = 0.50000   % new interval: [0, 0.50000]
c = 0.25000
    a = 0.25000   % new interval: [0.25000, 0.50000]
c = 0.37500
    a = 0.37500   % new interval: [0.37500, 0.50000]
c = 0.43750
    a = 0.43750   % new interval: [0.43750, 0.50000]
c = 0.46875
    b = 0.46875   % new interval: [0.43750, 0.46875]
c = 0.45312
    b = 0.45312   % new interval: [0.43750, 0.45312]
c = 0.44531
    a = 0.44531   % new interval: [0.44531, 0.45312]
c = 0.44922
    a = 0.44922   % new interval: [0.44922, 0.45312]
c = 0.45117
octave> f(c) ans = 6.4520e-04
```

[2]This is what gives the algorithm the alternative name "midpoint rule."

Algorithm 3 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 $f(b) \notin \mathbb{R}$ or $a + b \notin \mathbb{R}$ [overflow] then
      return ERROR
  end if
⋆START
  $i \leftarrow 0$
  $c \leftarrow \frac{a+b}{2}$ [both $NaN$ and $\infty$ can happen]
  while $|f(c)| \geq \epsilon$ and $i \leq N$ do
      [Never ever compare signs multiplying]
      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 (overflow possible)]
  end while
  if $i > N$ then
      return ERROR
  end if
  return $c$
```

As one can see, 0.45117 is an approximate root of $\cos(e^x)$, in the sense that $|\cos(e^{0.45117})| < 0.001$.

3. Newton-Raphson's Algorithm

A classical geometric idea (which also appears in approximation theory) is to use the *best*[3] *linear approximation* to f in order to compute an approximate solution to $f(x) = 0$. This best linear⋆ approximation is, of course, the tangent line to f, which is computed using the derivative $f'(x)$. So, instead of trying to solve $f(x) = 0$ directly, one draws the tangent to the graph of f at $(x, f(x))$ and finds the meeting

Examples

[3]From the point of view of infinitesimal analysis and polynomial approximation.

point of this line with the OX axis. Obviously, this will most likely not be a root of f but *in sufficiently general conditions*, it is expected to approach one. If the process is repeated enough times, it get nearer a root of f. This is the idea of Newton-Raphson's method.

Recall that the equation of the line passing through (x_0, y_0) with slope b is:

$$Y = b(X - x_0) + y_0$$

so that the equation of the tangent line to the graph of f at $(x_0, f(x_0))$ is (assuming f has a derivative at x_0):

$$Y = f'(x_0)(X - x_0) + f(x_0).$$

The meeting point between this line and OX is

$$(x_1, y_1) = \left(x_0 - \frac{f(x_0)}{f'(x_0)}, 0\right)$$

assuming it exists (i.e. $f'(x_0) \neq 0$).

If x_1 is not an approximate root of f with the desired precision, one proceeds in the same way at the point $(x_1, f(x_1))$. After having performed n steps, the next point x_{n+1} takes the form:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

One carries on until the desired precision or a bound in the number of iterations is reached. This is Algorithm 4. In its formal expression, we only specify a possible error place in order to keep it clear but one has to take into account that each time a computation is performed a floating-point error might occur.

EXAMPLE 8. Take the function $f(x) = e^x + x$, which is easily seen to have a unique zero on the whole real line (why is this so and why is it easy to see?). Its derivative is $f'(x) = e^x + 1$. Let us use $x_0 = 1$ as the seed. An implementation of Newton-Raphson's algorithm in Octave might show the following steps:

```
octave> c=newtonF(f, fp, 1, .0001, 5)
xn = 0
xn = -0.500000000000000
xn = -0.566311003197218
xn = -0.567143165034862
xn = -0.567143290409781
xn = -0.567143290409784
c = -0.567143290409784
octave> f(c)
ans = -1.11022302462516e-16
```

Algorithm 4 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
  $i \leftarrow 0$
  while $|f(x_i)| \geq \epsilon$ and $i \leq N$ do
      $x_{i+1} \leftarrow x_i - \frac{f(x_i)}{f'(x_i)}$
      $i \leftarrow i + 1$
  end while
  if $i > N$ then
      return ERROR
  end if
  $c \leftarrow x_i$
  return $c$
```

The convergence speed (which will be studied in detail in 5.5) is clear in the sequence of `xn`, as is the very good approximation to 0 of $f(c)$ for the approximate root after just 5 iterations.

EXAMPLE 9. However, the students are suggested to try Newton-Raphson's algorithm for the function $\cos(e^x)$ in Example 7. A strange phenomenon takes place: the algorithm is almost never convergent and no approximation to any root happens. Is there a simple explanation for this fact?

EXAMPLE 10. Newton-Raphson's method can behave quite strangely when $f'(c) = 0$, if c is a root of f. We shall develop some examples in the exercises and in the practicals.

4. The Secant Algorithm

Newton-Raphson's algorithm contains the evaluation of $\frac{f(x_n)}{f'(x_n)}$ for which one has to compute not only $f(x_n)$ but also $f'(x_n)$, which may be too costly. Moreover, there are cases in which one does not have true information on $f'(x)$, so that assuming it can be computed may be utopic.

The simplest solution of this problem is to approximate the value of the derivative using the geometric idea that "the tangent line is the limit of the secants"; instead of computing the tangent line one approximates it by means of two points near each other. Recall that the derivative of $f(x)$ at c is (if it exists) the limit

$$\lim_{h\to 0} \frac{f(c+h)-f(c)}{h}.$$

When applying Newton-Raphon's algorithm, if instead of using a single point x_n, one makes use of two: x_n and x_{n-1}, then the derivative $f'(x_n)$ can be approximated by means of

$$f'(x_n) \simeq \frac{f(x_n)-f(x_{n-1})}{x_n - x_{n-1}}, \tag{1}$$

so that the formula for computing x_{n+1} becomes, using this approximation,

$$x_{n+1} = x_n - f(x_n)\frac{x_n - x_{n-1}}{f(x_n)-f(x_{n-1})},$$

and the Secant Algorithm is obtained. Notice that, in order to start it, *two seeds* are required, instead of one. The coefficient of $f(x_n)$ in the iteration is just the reciprocal of approximation (1).

Algorithm 5 The Secant Algorithm.

```
  Input: A function $f(x)$, a tolerance $\epsilon > 0$, a bound for the number
  of iterations $N > 0$ and two seeds $x_{-1}, x_0 \in \mathbb{R}$
  Output: Either a real number $c \in \mathbb{R}$ with $|f(c)| < \epsilon$ or an error
  message
⋆START
  $i \leftarrow 0$
  while $|f(x_i)| \geq \epsilon$ and $i \leq N$ do
      $x_{i+1} \leftarrow x_i - f(x_i)\frac{x_i - x_{i-1}}{f(x_i)-f(x_{i-1})}$
      $i \leftarrow i+1$
  end while
  if $i > N$ then
      return ERROR
  end if
  $c \leftarrow x_i$
  return $c$
```

It is worthwhile, when implementing this method, keeping in memory not just x_n and x_{n-1} but also the values $f(x_n)$ and $f(x_{n-1})$ so as not to recompute them.

EXAMPLE 11. Take the same function as in example 8, that is $f(x) = e^x + x$. Let us use $x_0 = 1$ and $x_1 = 0.999$ as the two seeds. An implementation of the secant algorithm in Octave might show the following steps:

```
octave> c=secant(f, 1, 0.999, .0001, 6)
xn = -3.65541048223395e-04
xn = -0.368146813789761
xn = -0.544505751846815
xn = -0.566300946933390
xn = -0.567139827650291
c = -0.567139827650291
octave> f(c)
ans =  5.42664370639656e-06
```

It needs an iteration more than Newton-Raphson's method to achieve the same precision.

EXAMPLE 12. The secant algorithm has the same problem with the function of Example 7 because it is essentially an approximation to Newton-Raphson's, so that if this fails, it is expected for the secant method to fail as well.

5. Fixed Points

Fixed point algorithms —which, as we shall see later, include the previous ones indirectly— are based on the notion of *contractivity*, which reflects the idea that a function (a *transformation*) may map pairs of points in such a way that the images are always nearer than the original two (i.e. the function *shrinks*, *contracts* the initial space). This idea, linked to differentiability, leads to that of *fixed point* of an iteration and, by means of a little artifact, to the approximate solution of general equations using just iterations of a function.

5.1. Contractivity and equations $g(x) = x$.

Let g be a real-valued function of one real variable, differentiable at c. That is, for any infinitesimal o, there exists another one o_1 such that

$$g(c + o) = g(c) + g'(c)o + oo_1,$$

which means that *near c, the function g is very similar to its linear approximation.*

Assume that o is the width of a "small" interval centered at c. Removing the *supralinear* error (the term oo_1) for approximating, one might think that $(c - o, c + o)$ is mapped into $(g(c) - g'(c)o, g(c) + g'(c)o)$: that is, an interval of radius o is mapped into one of radius $g'(c)o$ (it dilates or shrinks by a factor of $g'(c)$). This is essentially what motivates the Jacobian Theorem in integration: the derivative measures the dilation or contraction which the real line undergoes at a point when transformed by g. If $|g'(c)| < 1$, the real line *contracts* near c. As a matter of fact, one usually reads the following

DEFINITION 8. A map $f : [a, b] \to \mathbb{R}$ is *contractive* if $|f(x) - f(y)| < |x - y|$ for any two $x, y \in [a, b]$.

Notice that if f is everywhere differentiable and $|f'(x)| < 1$ for all $x \in [a, b]$ then f is contractive.

Assume for simplicity that g' is continuous on $[a, b]$ and that $|g'(x)| < 1$ for all $x \in [a, b]$ —i.e., g "shrinks everywhere along $[a, b]$". There are several facts to notice. First of all, the conditions imply that there exists $\lambda < 1$ such that $|g'(x)| < \lambda$, by Weierstrass' Theorem applied to $g'(x)$. Moreover, by the Mean Value Theorem, for any $x_1, x_2 \in [a, b]$, the following inequality holds:

$$|g(x_1) - g(x_2)| \le \lambda |x_1 - x_2|,$$

which means: *the distance between the images of two arbitrary points is less than λ times the original distance between the points*, and, as $\lambda < 1$, what really happens is that *the distance between the images is always less than the original distance*. That is, the map g is *shrinking* the interval $[a, b]$ everywhere. It is as if $[a, b]$ were a piece of cotton cloth and it was being washed with warm water: it shrinks everywhere.

The last paragraph contains many emphasis but they are necessary for understanding the final result: the existence of a fixed point, a point which gets mapped to itself.

From all the explanation above, one infers that the width of $g([a, b])$ is less than or equal to $\lambda(b - a)$. If moreover $g([a, b]) \subset [a, b]$ (that is, if g mapped $[a, b]$ into itself) then one could also compute $g(g([a, b]))$, which would be at most of width $\lambda\lambda(b - a) = \lambda^2(b - a)$. Now one could repeat the iteration indefinitely and $g \circ g \circ g \circ \cdots \circ g = g^{\circ n}([a, b])$ would have width less than $\lambda^n(b - a)$. As $\lambda < 1$, this width tends to 0 and using the Nested Intervals Principle, there must be a point $\alpha \in [a, b]$ for which $g(\alpha) = \alpha$. This is, by obvious reasons, a *fixed point*. Even more, the fact that $\lambda < 1$ implies that α is unique. There is one and only one fixed point for g in $[a, b]$. We have just given a plausible argument for the next result:

THEOREM 1. *Let $g : [a, b] \to [a, b]$ 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 $x \in [a, b]$, $|g'(x)| \le \lambda$, then there exists one and only one $\alpha \in [a, b]$ for which $g(\alpha) = \alpha$. Even more, for any $x_0 \in [a, b]$, if one defines*

$$x_n = g(x_{n-1}) \text{ for } n > 0$$

then

$$\lim_{n\to\infty} x_n = \alpha.$$

This means that under suitable conditions, the equation $g(x) = x$ has a single solution in the interval $[a, b]$. The explanation before the statement shows how to approximate this solution: take any $x \in [a, b]$ and compute $g(x)$, $g(g(x))$, ... The limit (which exists under those conditions) is always α, regardless of x.

Hence,

solving equations of the form $g(x) = x$ for contractive g is utterly simple: **just iterate** g.

5.2. Application to General Equations $f(x) = 0$**.** In real life (if this happens to relate to real life at all, anyway), nobody comes across an equation of the form $g(x) = x$. One always encounters problems like $f(x) = 0$ or, more generally $f(x) = c$, c being a constant, which are easily turned into $f(x) = 0$.

But this is not a problem because

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

so that, *searching for a root of $f(x)$ is the same thing as searching for a fixed point of $g(x) = f(x) + x$*. Or, for the same reason, of $x - f(x)$.

REMARK (Skip in a first reading). As a matter of fact, if $\phi(x)$ is a nowhere zero function, then *searching for a root of $f(x)$ is the same as searching for a fixed point of $g(x) = x - \phi(x)f(x)$*. This allows, for example, to *scale* f so that its derivative is approximately 1 and g' is small in order to accelerate convergence. Or one can just take $g(x) = x - cf(x)$ for a suitable c which makes the derivative of g relatively small in absolute value.

5.3. The Algorithm. This is probably the easiest algorithm to implement, as it only requires computing the value of g each time. As any other, it requires a tolerance ϵ and a maximum number of iterations N. The drawback is that the algorithm can be useless if $[a, b]$ is not mapped into itself. *This has to be checked beforehand.*

REMARK 1. Let $g : [a, b] \to \mathbb{R}$ be a map. If a fixed point of g in $[a, b]$ is to be found using contractivity, it is necessary:

- If g is differentiable[4] in $[a, b]$, there must exist $\lambda \in \mathbb{R}$ such that $0 < \lambda < 1$ and for which $|g'(x)| \le \lambda$ for all $x \in [a, b]$.

[4]If g it is not, then the requirement —the Lipschitz condition— is much harder to test.

Assumed both checks have been performed, the method for finding a fixed point of $g : [a, b] \to [a, b]$ is stated in Algorithm 6:

Algorithm 6 Fixed Point.

```
  Input: A function $g$ (contractive etc...), a seed $x_0 \in [a, b]$, a tolerance $\epsilon > 0$ and a maximum number of iterations $N > 0$
  Output: either $c \in [a, b]$ such that $|c - g(c)| < \epsilon$ or an error message
⋆START
  $i \leftarrow 0, c \leftarrow x_0$
  while $|c - g(c)| \geq \epsilon$ and $i \leq N$ do
      $c \leftarrow g(c)$
      $i \leftarrow i + 1$
  end while
  if $i > N$ then
      return ERROR
  end if
  return $c$
```

EXAMPLE 13. For the fixed point algorithm we shall use the same function as for the bisection method, that is $f(x) = \cos(e^x)$. In order to find a root of this function, we need to turn the equation

$$\cos(e^x) = 0$$

into a fixed-point problem. This is always done in the same (or similar) way: the above equation is obviously equivalent to

$$\cos(e^x) + x = x,$$

which is a fixed-point problem. Let us call $g(x) = \cos(e^x) + x$, whose fixed point we shall try to find.

In order to apply the fixed-point theorem, one needs an interval $[a, b]$ which is mapped into itself. It is easily shown that $g(x)$ decreases near $x = 0.5$ and, as a matter of fact, that it does so in the interval $I = [0.4, 0.5]$. Moreover, $g(0.4) \simeq 0.478+$ while $g(0.5) \simeq 0.4221+$, which implies that the interval I is mapped into itself by g (this is probably the most difficult part of a fixed-point problem: finding an appropriate interval which gets mapped into itself). The derivative of g is $g'(x) = -e^x \sin(e^x) + 1$, whose absolute value is, in that interval, less than 0.8 (it is less than 1 in the whole interval $[0, 1]$, as a matter of fact). This is the second condition to be verified in order to apply the theorem.

Now we can be certain that there is a fixed point for $g(x)$ in $[0.4, 0.5]$. Matlab would give a sequence like

```
octave> c=fixed_point(g, 0.5, .0001, 10)
xn =  0.422153896052972
xn =  0.467691234530252
xn =  0.442185884541927
xn =  0.456876713444163
xn =  0.448538950613744
xn =  0.453312782246289
xn =  0.450592834887592
xn =  0.452146949739883
xn =  0.451260386650212
xn =  0.451766601977469
c =  0.451766601977469
octave> f(c)
ans = -2.88890800240215e-04
```

The convergence speed does not look too good, which is usual in fixed-point problems.

REMARK (Skip in first reading). The fixed point algorithm can be used, as explained in 5.2, for finding roots of general equations using a suitable factor; to this end, if the equation is $f(x) = 0$, one can use any function $g(x)$ of the form

$$g(x) = x - kf(x)$$

where $k \in \mathbb{R}$ is an adequate number. This transformation is performed so that the derivative of g is around 0 near the root and so that if c is the (unknown) root, then g defines a contractive map on an interval of the form $[c - \rho, c + \rho]$ (which will be the $[a, b]$ used in the algorithm).

The hard part is to check that g is a contractive map of $[a, b]$ onto itself.

5.4. Convergence Speed of the Fixed Point Method. If the absolute value of the derivative is bounded by $\lambda < 1$, the convergence speed of the fixed point algorithm is easily bounded because $[a, b]$ is mapped onto a sub-interval of width at most $\lambda(b - a)$ and this contraction is repeated at each step. So, after i iterations, the width of the image set is at most $\lambda^i(b - a)$. If $\lambda < 10^{-1}$ and $(b - a) < 1$, for example, then one can guarantee that each iteration gives an extra digit of precision in the root. However, λ tends to be quite large (like 0.9) and the process is usually slow.

EXAMPLE 14. If $[a, b] = [0, 1]$ and $|g'(x)| < 0.1$, then after each iteration there is an exact decimal digit more in x as an approximation to the fixed point c, regardless of the initial value of the seed x.

5.5. Newton-Raphson's convergence speed. As a matter of fact, Newton-Raphson's algorithm, under suitable conditions, is just a fixed point algorithm and the analysis of the convergence speed for these can be applied to it in this case. Notice that the expression:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

corresponds to the search of a fixed point of

$$g(x) = x - \frac{1}{f'(x)}f(x)$$

which, as explained above, is a way of turning the equation $f(x) = 0$ into a fixed-point problem. The derivative of g is, in this case:

$$g'(x) = 1 - \frac{f'(x)^2 - f(x)f''(x)}{f'(x)^2}$$

which, at a root c of f (that is, $f(c) = 0$) gives $g'(c) = 0$. That is, the derivative of g is 0 at the fixed point of g. This makes convergence *very fast* when x is near c (and other conditions hold).

In fact, the following strong result (which is usually stated as "Newton-Raphson has quadratic convergence") can be proved:

THEOREM 2. *Assume f is a twice-differentiable function on an interval $[r-\epsilon, r+\epsilon]$, that r is a root of f and that*

- *The second derivative of f is bounded from above: $|f''(x)| < K$ for $x \in [r-\epsilon, r+\epsilon]$,*
- *The first derivative of f is bounded from below: $|f'(x)| > L > 0$ for $x \in [r-\epsilon, r+\epsilon]$*

then, if $x_k \in [r-\epsilon, r+\epsilon]$, the following term in Newton-Raphson's iteration is also in that interval and

$$|r - x_{k+1}| < |\frac{K}{2L}||r - x_k|^2.$$

And, as a corollary:

COROLLARY 1 (Duplication of exact digits in Newton-Raphson's metod). *Under the conditions of Theorem 2, if $\epsilon < 0.1$ and $K < 2L$ then, for $n \geq k$, each iteration x_{n+1} of Newton-Raphson approximates c with twice the number of exact digits as x_n.*

PROOF. This happens because if $k = 0$ then x_0 has at least one exact digit. By the Theorem, $|x_1 - r|$ is less than $0.1^2 = .01$. And from this point on, the number of zeroes in the decimal expansion of $|x_n - r|$ gets doubled each time. □

In order to use Theorem 2 or its corollary, one has to:

- Be certain that a root is near. The usual way to check this is using Bolzano's Theorem (i.e. checking that f changes sign on a sufficiently small interval).
- Bound the width of the interval (for example, by $1/10$).
- Bound $f''(x)$ from above and $f'(x)$ from below on the interval (this is usually the hardest part).

6. Annex: Matlab/Octave Code

Some code with "correct" implementations of the algorithms of this chapter are included here. They should run both on Matlab and Octave. However, notice that the code here *does not check for floating-point exceptions*. For example, if one uses the function $\log(x)$ with any of this programs and it is evaluated at a negative number, the programs will happily continue but the results will probably be either absurd or not real. The decision not to include those checks has been made for simplicity.

6.1. The Bisection Algorithm.

The code in Listing 2.1 implements the bisection algorithm for Matlab/Octave. The input parameters are:

f: and anonymous function,
a: the left endpoint of the interval,
b: the right endpoint of the interval,
epsilon: a tolerance (by default, `eps`),
n: a bound for the number of iterations (by default, 50).

The output may be null (if there is no sign change, with an appropriate message) or an approximate root (to the tolerance) or a warning message together with the last computed value (if the tolerance has not been reached in the specified number of iterations). The format of the output is a pair `[z, N]` where `z` is the approximate root (or the last computed value) and `N` is the number of iterations performed.

```
% Bisection Algorithm with tolerance and stopping condition
% Notice that at any evaluation f(...) an error might take place
% which is NOT checked.
function [c, N] = Bisec(f, a, b, epsilon = eps, n = 50)
  N = 0;
  if(f(a)*f(b)>0)
    warning('no sign change')
    return
  end
  % store values in memory
  fa = f(a);
  fb = f(b);
  if(fa == 0)
    c = a;
    return
  end
  if(fb == 0)
    c = b;
    return
  end
  c = (a+b)/2
  fc = f(c);
  while(abs(fc) >= epsilon & N < n)
    N = N + 1;
    % multiply SIGNS, not values
    if(sign(fc)*sign(fa) < 0)
      b = c;
      fb = fc;
    else
      a = c;
      fa = fc;
    end
    % An error might happen here
    c = (a+b)/2;
    fc = f(c);
  end
  if(N >= n)
    warning("Tolerance not reached.")
  end
end
```

LISTING 2.1. Code for the Bisection Algorithm.

6.2. Newton-Raphson's Algorithm. Newton-Raphson's algorithm is easier to implement (always without checking for floating point exceptions) but requires more complex input data: the derivative of f, another anonymous function. The code in Listing 2.2 does not perform symbolic computations so that this derivative must be provided by the user.

The input is, then:

`f`: an anonymous function,
`fp`: another anonymous function, the derivative of `f`,
`x0`: the seed,
`epsilon`: the tolerance (by defect `eps`),
`N`: the maximum number of iterations (by default 50).

The format of the output, in order to facilitate its study, is a pair `[xn,N]`, where `xn` is the approximate root (or the last computed value) and `N` is the number of iterations performed.

```
% Newton-Raphson implementation
function [z n] = NewtonF(f, fp, x0, epsilon = eps, N = 50)
  n = 0;
  xn = x0;
  % 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.');
  end
end
```

LISTING 2.2. Code for Newton-Raphson's Algorithm

CHAPTER 3

Numerical Solutions to Linear Systems of Equations

Some of the main classical algorithms for (approximately) solving systems of linear equations are discussed in this chapter. We begin with Gauss' reduction method (and its interpretation as the *LU* factorization). The notion of *condition number* of a matrix and its relation with relative error is introduced (in a very simplified way, without taking into account the errors in the matrix). Finally, the fixed-point algorithms are introduced and two of them (Jacobi's and Gauss-Seidel's) are explained.

All three algorithms are relevant by themselves but knowing the scope of each one is as important, at least from the theoretical standpoint; for example, the requirement that the matrix be *convergent* (the analogue to contractivity). We shall not speak about the spectrum nor use the eigenvalues; these are of the utmost importance but the students have no true knowledge of them. We prefer to insist on the necessity of convergence (and the student will be able to apply it to each case when need comes).

In this chapter, the aim is to (approximately) solve a system of linear equations of the form

$$Ax = b \tag{2}$$

where A is a *square* matrix of order n and b is a vector in $\mathbb{R}^n$. We always assume unless explicitely stated that A *is non-singular*, so that the system is consistent and has a unique solution.

1. Gauss' Algorithm and *LU* Factorization

Starting from system (2), Gauss' method transforms A and b by means of "simple" operations into an upper triangular matrix $\tilde{A}$ and a vector $\tilde{b}$ such that the new system $\tilde{A}x = \tilde{b}$ is solvable by *regressive substitution* (i.e. x_n is directly solvable and each variable x_k is solvable in terms of $x_{k+1}, \dots, x_n$). Obviously, in order that this transformation make sense, the main requirement is that the solution to the new system be the same as that of the original[1]. To this end, only the following operation is permitted:

- Any equation E_i (the $i-$th row of A together with the $i-$th element of b) may be substituted by a linear combination of the form $E_i + \lambda E_k$ for some $k < i$ and $\lambda \in \mathbb{R}$. In this case, b_i is substituted with $b_i + \lambda b_k$.

The fact that E_i appears with coefficient 1 in the substituting expression ($E_i + \lambda E_k$) is what guarantees that the new system has the same solution as the original one.

Let $\overline{A}$ be the augmented matrix $\overline{A} = (A|b)$.

LEMMA 1. *In order to transform a matrix $\overline{A}$ into a matrix $\tilde{A}$ using the above operation, it is enough to multiply $\overline{A}$ on the left by the matrix $L_{ik}(\lambda)$ whose elements are:*

- *If $m = n$, then $(L_{ik}(\lambda))_{mn} = 1$ (diagonal with 1).*
- *If $m = i, n = k$, then $(L_{ik}(\lambda))_{mn} = \lambda$ (the element (i,k) is λ).*
- *Any other element is 0.*

EXAMPLE 15. Starting with the following $\overline{A}$

$$\overline{A} = \left(\begin{array}{cccc|c} 3 & 2 & -1 & 4 & -1 \\ 0 & 1 & 4 & 2 & 3 \\ 6 & -1 & 2 & 5 & 0 \\ 1 & 4 & 3 & -2 & 4 \end{array}\right)$$

and combining row 3 with row 1 times -2 (in order to "make a zero at the 6"), then one has to multiply by $L_{31}(-2)$

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -2 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 3 & 2 & -1 & 4 & -1 \\ 0 & 1 & 4 & 2 & 3 \\ 6 & -1 & 2 & 5 & 0 \\ 1 & 4 & 3 & -2 & 4 \end{pmatrix} = \begin{pmatrix} 3 & 2 & -1 & 4 & -1 \\ 0 & 1 & 4 & 2 & 3 \\ 0 & -5 & 4 & -3 & 2 \\ 1 & 4 & 3 & -2 & 4 \end{pmatrix}.$$

Algorithm 7 is a simplified statement of Gauss' reduction method. The line with a comment `[*] 7` is precisely the multiplication of $\tilde{A}$ on the left by $L_{ji}(-m_{ji})$. In the end, $\tilde{A}$, which is upper triangular, is a product of these matrices:

$$\tilde{A} = L_{n,n-1}(-m_{n,n-1})L_{n,n-2}(-m_{n,n-2}) \cdots L_{2,1}(-m_{2,1})A = \tilde{L}A \tag{3}$$

[1]However, it is important to realize that no numerical algorithm intends to find the *exact* solutions to any equation or system of equations. So this "sameness" of the solutions relates only to the *theoretical* algorithm

and $\tilde{L}$ is a lower triangular matrix with 1's on the diagonal (this is a simple exercise to verify). It is easy to check (although this looks a bit like *magic*) that

LEMMA. *The inverse matrix of $\tilde{L}$ in (3) is the lower triangular matrix whose (i, j) entry is m_{ij}. That is,*

$$(L_{n,n-1}(-m_{n,n-1})L_{n,n-2}(-m_{n,n-2})\cdots L_{2,1}(-m_{2,1}))^{-1} = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ m_{21} & 1 & 0 & \dots & 0 \\ m_{31} & m_{32} & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ m_{n1} & m_{n2} & m_{n3} & \dots & 1 \end{pmatrix}$$

Algorithm 7 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 an operation on a row]
          $\tilde{A}_j \leftarrow \tilde{A}_j - m_{ji}\tilde{A}_i$   [⋆]
          $\tilde{b}_j \leftarrow \tilde{b}_j - m_{ji}\tilde{b}_i$
          $j \leftarrow j + 1$
      end while
      $i \leftarrow i + 1$
  end while
  return $\tilde{A}, \tilde{b}$
```

We have thus proved the following result:

THEOREM 3. *If in the Gauss' reduction process no element of the diagonal is* 0 *at any step, then there is a lower triangular matrix L whose elements are the corresponding coefficients at their specific place and an upper triangular matrix U such that*

$$A = LU$$

and such that the system $Ux = \tilde{b} = L^{-1}b$ is equivalent to the initial one $Ax = b$.

With this result, a factorization of A is obtained which simplifies the resolution of the original system, as $Ax = b$ can be rewritten $LUx = b$ and one can proceed step by step:

- First the system $Ly = b$ is solved by *direct substitution* —that is, from top to bottom, without even dividing.
- Then the system $Ux = y$ is solved by *regressive substitution* —from bottom to top; divisions will be needed at this point.

This solution method only requires storing the matrices L and U in memory and is very fast.

1.1. Pivoting Strategies and the *LUP* Factorization. If during Gaussian reduction a pivot appears (the element which determines the multiplication) of value approximately 0, either the process cannot be continued or one should expect large errors to take place, due to rounding and truncation[2]. This problem can be tackled by means of *pivoting strategies*, either swapping rows or both rows and columns. If only rows are swapped, the operation is called *partial pivoting*. If swapping both rows and columns is allowed, *total pivoting* takes place.

DEFINITION 9. A *permutation matrix* is a square matrix whose entries are all 0 but for each row and column, in each of which there is exactly a 1.

Permutation matrices can be built from the identity matrix by *permuting* rows (or columns).

A permutation matrix P has, on each row, a single 1 and the rest entries are 0. If on row i the only 1 is on column j, the multiplication PA means "swap rows j and i of A."

Obviously, the determinant of a permutation matrix is not zero (it is either 1 or -1). It is not so easy to show that the inverse of a

[2]Recall Example 5 of Chapter 1, where a tiny truncation error in a denominator became a huge error in the final result.

permutation matrix is another permutation matrix and, as a matter of fact, that $P^{-1} = P^T$ (the transpose) if P is a permutation matrix.

LEMMA 2. *If A is an $n \times m$ matrix and P is a permutation matrix of order n having only 2 non-zero elements out of the diagonal, say (i, j) and (j, i), then PA is the matrix obtained from A by swapping* rows *i and j. On the other hand, if P is of order m, then AP is the matrix A with* columns *i and j swapped.*

DEFINITION 10. A *pivoting strategy* is followed in Gauss' reduction process if the pivot element chosen at step i is the one with greatest absolute value.

In order to perform Gauss' algorithm with partial pivoting, one can just follow the same steps but choosing, at step i, the row $j > i$ in which the pivot has the greatest absolute value and swapping rows i and j.

Pivoting strategies give rise to a factorization which differs from LU in that permutation matrices are allowed as factors.

LEMMA 3. *Given the linear system of equations $Ax = b$ with A non-singular, there is a permutation matrix P and two matrices L, U, the former lower triangular (with only 1 on the diagonal) and the latter upper triangular such that*

$$PA = LU.$$

This result is proved indirectly by recurrence (we are not going to do it).

Both Matlab and Octave include the function `lu` which, given A, returns three values: L, U and P.

For example, if

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ -1 & -2 & 5 & 6 \\ -1 & -2 & -3 & 7 \\ 0 & 12 & 7 & 8 \end{pmatrix}$$

then

$$L = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ -1 & 0 & 0 & 1 \end{pmatrix} \quad U = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 0 & 12 & 7 & 8 \\ 0 & 0 & 8 & 10 \\ 0 & 0 & 0 & 11 \end{pmatrix}, \quad P = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

In order to compute L, U and P one only needs to follow Gauss' algorithm and *whenever a swap of rows i and j is performed*, the same swap has to be performed on *L up to the $i - 1$ column* (that is, if a swap is needed on A at step i then the rows i and j are swapped *but only the columns* 1 *to* $i-1$ are involved). Also, the P computed up to step i has to be multiplied on the left by P_{ij} (the permutation matrix for i and j).

This can be stated as Algorithm 8.

Algorithm 8 *LUP* Factorization for a matrix A

```
  Input: A matrix $A$ of order $n$
  Output: Either an error or three matrices: $L$ lower triangular with
  1 on the diagonal, $U$ upper triangular and a permutation matrix $P$
  such that $LU = PA$
  Comment: $P_{ip}$ is the permutation matrix permuting rows $i$ and $p$.
⋆START
  $L \leftarrow \mathrm{Id_n}, \mathrm{U} \leftarrow \mathrm{Id_n}, \mathrm{P} \leftarrow \mathrm{Id_n}, i \leftarrow 1$
  while $i < n$ do
      $p \leftarrow$ row index such that $|U_{pi}|$ is maximum, with $p \geq i$
      if $U_{pi} = 0$ then
          return ERROR [division by zero]
      end if
      [swap rows $i$ and $p$]
      $P \leftarrow P_{ip}P$
      $U \leftarrow P_{ip}U$
      [on $L$ swap only rows $i$ and $p$ on the submatrix
      $n \times (i-1)$ at the left, see the text]
      $L \leftarrow \tilde{L}$
      [combine rows on $U$ and keep track on $L$]
      $j \leftarrow i+1$
      while $j <= n$ do
          $m_{ji} \leftarrow U_{ji}/U_{ii}$
          $U_j \leftarrow U_j - m_{ij}U_i$
          $L_{ji} \leftarrow m_{ji}$
          $j \leftarrow j+1$
      end while
      $i \leftarrow i+1$
  end while
  return $L, U, P$
```

2. Condition Number: behavior of the relative error

We now deal briefly with the question of the *stability* of the methods for solving systems of linear equations. This means: if instead of starting with a vector b (which might be called the "initial condition"), one starts with a slight modification $\tilde{b}$, how much does the solution to the new system differ from the original one? The proper way to ask this question uses the *relative* error, not the absolute one. Instead of system (2), consider the modified one

$$Ay = b + \delta_b$$

where δ_b is a *small vector*. The new solution will have the form $x + \delta_x$, for some δ_x (which one *expects* to be small as well).

EXAMPLE 16. Consider the system of linear equations

$$\begin{pmatrix} 2 & 3 \\ 4 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \end{pmatrix}$$

whose solution is $(x_1, x_2) = (0.1, 0.6)$. If we take $\tilde{b} = (2.1, 1.05)$ then $\delta_b = (0.1, 0.05)$ and the solution of the new system

$$\begin{pmatrix} 2 & 3 \\ 4 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 2.1 \\ 1.05 \end{pmatrix}$$

is $(x_1, x_2) = (0.105, 0.63)$ so that $\delta_x = (0.005, 0.03)$. In this case, a small increment δ_b gives rise to a small increment δ_x but this needs not be so.

However, see Example 17 for a system with a very different behaviour.

The size of vectors is measured using *norms* (the most common one being *length*, in Euclidean space, but we shall not make use of this one). As x is a solution, one has

$$A(x + \delta_x) = b + \delta_b,$$

so that

$$A\delta_x = \delta_b,$$

but, as we want to compare δ_x with δ_b, we can write

$$\delta_x = A^{-1}\delta_b$$

so that, taking *sizes* (i.e. *norms*, which are denoted with $\|\,\|$), we get

$$\|\delta_x\| = \|A^{-1}\delta_b\|.$$

Recall that we are trying to asses the *relative displacement*, not the absolute one. To this end we need to include $\|x\|$ in the left hand side of the equation. The available information is that $Ax = b$, from which $\|Ax\| = \|b\|$. Hence,

$$\frac{\|\delta_x\|}{\|Ax\|} = \frac{\|A^{-1}\delta_b\|}{\|b\|}, \tag{4}$$

but this is not very useful (it is obvious, as it stands). However, assume there is some kind of *matrix norm* $\|A\|$ which satisfies that $\|Ax\| \leq \|A\|\|x\|$. Then, performing this substitution in Equation (4), one gets

$$\frac{\|A^{-1}\delta_b\|}{\|b\|} = \frac{\|\delta_x\|}{\|Ax\|} \geq \frac{\|\delta_x\|}{\|A\|\|x\|}$$

and, applying the same reasoning to the right hand side of (4), one gets

$$\frac{\|A^{-1}\|\|\delta_b\|}{\|b\|} \geq \frac{\|A^{-1}\delta_b\|}{\|b\|},$$

and combining everything,

$$\frac{\|A^{-1}\|\|\delta_b\|}{\|b\|} \geq \frac{\|\delta_x\|}{\|A\|\|x\|}$$

so that, finally, one gets an upper bound for the *relative displacement of x*:

$$\frac{\|\delta_x\|}{\|x\|} \leq \|A\|\|A^{-1}\|\frac{\|\delta_b\|}{\|b\|}$$

The "matrix norm" which was mentioned above does exist. In fact, there are many of them. We shall only make use of the following one in these notes:

DEFINITION 11. The *infinity norm* of a square matrix $A = (a_{ij})$ is the number

$$\|A\|_\infty = \max_{1\leq i\leq n} \sum_{j=1}^{n} |a_{ij}|,$$

that is, the maximum of the sums of absolute values of each row.

The following result relates the infinity norms of matrices and vectors:

LEMMA 4. *The* infinity norm *is such that, for any vector x, $\|Ax\|_\infty \leq \|A\|_\infty\|x\|_\infty$, where $\|x\|_\infty$ is the norm given by the maximum of the absolute values of the coordinates of x.*

This means that, if one measures the size of a vector by its largest coordinate (in absolute value), and one calls it $\|x\|_\infty$, then

$$\frac{\|\delta_x\|_\infty}{\|x\|_\infty} \leq \|A\|_\infty \|A^{-1}\|_\infty \frac{\|\delta_b\|_\infty}{\|b\|_\infty}.$$

The product $\|A\|_\infty \|A^{-1}\|_\infty$ is called the *condition number of A for the infinity norm*, is denoted $\kappa(A)$ and is a bound for the maximum possible displacement of the solution when the initial vector gets displaced. The greater the condition number, the greater (to be expected) the displacement of the solution when the initial condition (independent term) changes a little.

The condition number also bounds *from below* the relative displacement:

LEMMA 5. *Let A be a nonsingular matrix of order n and x a solution of $Ax = b$. Let δ_b be a "displacement" of the initial conditions and δ_x the corresponding "displacement" in the solution. Then:*

$$\frac{1}{\kappa(A)} \frac{\|\delta_b\|}{\|b\|} \leq \frac{\|\delta x\|}{\|x\|} \leq \kappa(A) \frac{\|\delta_b\|}{\|b\|}.$$

So that the relative displacement (or error) can be bounded using the relative residue *(the number $\|\delta_b\| / \|b\|$).*

The following example shows how large condition numbers are usually an indicator that solutions may be strongly dependent on the initial values.

EXAMPLE 17. Consider the system

$$\begin{aligned} 0.853x + 0.667y &= 0.169 \\ 0.333x + 0.266y &= 0.067 \end{aligned} \tag{5}$$

whose condition number for the infinity norm is 376.59, so that a relative change of a thousandth of unit in the initial conditions (vector b) is expected to give rise to a relative change of more than 37% in the solution. The exact solution of (5) is $x = 0.055+, y = 0.182+$.

However, the size of the condition number is a tell-tale sign that a small perturbation on the system will modify the solutions greatly. If, instead of $b = (0.169, 0.067)$, one uses $b = (0.167, 0.067)$ (which is a relative displacement of just 1.1% in the first coordinate), the new solution is $x = -0.0557+, y = 0.321+$, for which x has not even the same sign as in the original problem and y is displaced 76% from its original value. This is clearly unacceptable. If the equations describe a static system, for example, and the coefficients have been measured with up to 3 significant digits, then the system of equations *is useless*, as one cannot be certain that the measuring errors are meaningful.

3. Fixed Point Algorithms

Gauss' reduction method is a first attempt at a fast algorithm for solving linear systems of equations. It has two drawbacks: it is quite complex (in the technical sense, which means it requires *a lot of operations* to perform) and it depends greatly on the accuracy of divisions (and when small divisors do appear, large relative errors are expected to come up). However, the first issue (complexity) is the main one. A rough approximation shows that for a system with n variables, Gauss' algorithm requires approximately n^3 operations. For n in the tens of thousands this soon becomes impractical.

Fixed point algorithms tackle this problem by not trying to find an "exact" solution (which is the aim of Gauss' method) but to approximate one using the techniques explained in Chapter 2.

Start with a system like (2), of the form $Ax = b$. One can transform it into a fixed point problem by means of a "decomposing" of matrix A into two: $A = N - P$, where N is some invertible matrix. This way, $Ax = b$ can be written $(N - P)x = b$, i.e.

$$(N - P)x = b \Rightarrow Nx = b + Px \Rightarrow x = N^{-1}b + N^{-1}Px.$$

If one calls $c = N^{-1}b$ and $M = N^{-1}P$, then one obtains the following fixed point problem:

$$x = Mx + c$$

which can be solved (if at all) in the very same way as in Chapter 2: start with a seed x_0 and iterate

$$x_n = Mx_{n-1} + c,$$

until a sufficient precision is reached.

In what follows, the infinity norm $\|\ \|_\infty$ is assumed whenever the concept of "convergence" appears[3].

One needs the following results:

THEOREM 4. *Assume M is a matrix of order n and that $\|M\|_\infty < 1$. Then the equation $x = Mx + c$ has a unique solution for any c and the iteration $x_n = Mx_{n-1} + c$ converges to it for any initial value x_0.*

[3]However, all the results below apply to any matrix norm.

THEOREM 5. *Given M with $\|M\|_\infty < 1$, and given a seed x_0 for the iterative method of Theorem 4, if s is the solution to $x = Mx + c$, then the following bound holds:*

$$\|x_n - s\|_\infty \leq \frac{\|M\|_\infty^n}{1 - \|M\|_\infty} \|x_1 - x_0\|_\infty.$$

Recall that for vectors, the infinity norm $\|x\|_\infty$ is the maximum of the absolute values of the coordinates of x.

We now proceed to explain the two basic iterative methods for solving linear systems of equations: Jacobi's and Gauss-Seidel. Both rely on "decomposing" A in different ways: Jacobi takes N as the diagonal of A and Gauss-Seidel as the lower triangular part of A, including its diagonal.

3.1. Jacobi's Algorithm. If each coordinate x_i is solved explicitly in terms of the others, for the system $Ax = b$, then something like what follows appears:

$$x_i = \frac{1}{a_{ii}}(b_i - a_{i1}x_1 - \cdots - a_{ii-1}x_{i-1} - a_{ii+1}x_{i+1} - \cdots - a_{in}x_n),$$

in matrix form,

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

where D is the diagonal matrix whose nonzero elements are the diagonal of A. One can write,

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

which is a fixed point problem.

If $D^{-1}(A - D)$ satisfies the conditions of Theorem 4, then the iterations corresponding to the above expression converge for any choice of seed x_0 and the bound of Theorem 5 holds. This is called Jacobi's method. In order to verify the necessary conditions, the computation of $D^{-1}(A - D)$ is required, although there are other sufficient conditions (especially Lemma 6).

3.2. Gauss-Seidel's Algorithm. If instead of using the diagonal of A one takes its lower triangular part (including the diagonal) then one gets a system of the form

$$x = T^{-1}b - T^{-1}(A - T)x,$$

which is also a fixed point problem.

If $T^{-1}(A - T)$ satisfies the conditions of Theorem 4, then the corresponding iteration (called the Gauss-Seidel iteration) converges for any initial seed x_0 and the bound of Theorem 5 holds. In order to verify the conditions one should compute $T^{-1}(A-T)$, but there are other sufficient conditions.

3.3. Sufficient conditions for convergence. We shall use two results which guarantee convergence for the iterative methods explained above. One requires a technical definition, the other applies to positive definite matrices.

DEFINITION 12. A matrix $A = (a_{ij})$ is *strictly diagonally dominant* by rows if

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}|$$

for any i from 1 to n. That is, if the elements on the diagonal are greater in absolute value than the sum of the rest of the elements of their rows in absolute value.

For these matrices, the convergence of both Jacobi's and Gauss-Seidel's methods is guaranteed:

LEMMA 6. *If A is a* strictly diagonally dominant *matrix by rows then both Jacobi's and Gauss-Seidel's methods converge for any system of the form $Ax = b$ and any seed.*

For Gauss-Seidel, the following also holds:

LEMMA 7. *If A is a positive definite symmetric matrix, then Gauss-Seidel's method converges for any system of the form $Ax = b$.*

4. Annex: Matlab/Octave Code

Code for some of the algorithms explained is provided; it should work both in Matlab and Octave.

4.1. Gauss' Algorithm without Pivoting. Listing 3.1 implements Gauss' reduction algorithm for a system $Ax = b$, returning L, U and the new b vector, *assuming the multipliers are never* 0; if some is 0, then the process terminates with a message.

Input:

- **`A`:** a square matrix (if it is not square, the output gives the triangulation under the principal diagonal),
- **`b`:** a vector with as many rows as `A`.

The output is a trio `[L, At, bt]`, as follows:

- **`L`:** the lower triangular matrix (with the multipliers),
- **`At`:** the transformed matrix (which is U in the LU-factorization) and which is upper triangular.

bt: the transformed initial vector.

The new system to be solved is, hence, $At \times x = bt$.

```
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
  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(l,k) means ROW l, COLUMN k
    while(l<=n)
      L(l,k)=At(l,k)/At(k,k);
      % Combining rows is easy in Matlab
      At(l,k:n) = [0 At(l,k+1:n) - L(l,k) * At(k,k+1:n)];
      bt(l)=bt(l)-bt(k)*L(l,k);
      l=l+1;
    end
    k=k+1;
  end
end
```

LISTING 3.1. Gauss' Reduction Algorithm

4.2. *LUP* Factorization. Gauss' Reduction depends on the non-appearing of zeros on the diagonal and it can also give rise to large rounding errors if pivots are small. Listing 3.2 implements LUP factorization, which provides matrices L, U and P such that $LU = PA$, L and U being respectively lower and upper triangular and P a permutation matrix. Its input is

A: a square matrix of order n.
b: An n-row vector.

The output is a vector `[L, At, P, bt]` where, `L`, `At`, `P` and `bt`, are three matrices and a vector corresponding to L, U, P and the transformed vector $\tilde{b}$, according to the algorithm.

```
function [L, At, P, bt] = gauss_pivotaje(A,b)
  n = size(A);
  m = size(b);
  if(n(2) ~= m(1))
    warning('Dimensions of A and b do not match');
    return;
  end
  At=A;
  bt=b;
  L=eye(n);
  P=eye(n);
  i=1;
  while (i<n)
    j=i+1;
    % beware nomenclature:
    % L(j,i) is ROW j, COLUMN i
    % the pivot with greatest absolute value is sought
    p = abs(At(i,i));
    pos = i;
    for c=j:n
      u = abs(At(c,i));
      if(u>p)
        pos = c;
        p = u;
      end
    end
    if(u == 0)
      warning('Singular system');
      return;
    end
    % Swap rows i and p if i != p
    % in A and swap left part of  L
    % This is quite easy in Matlab, there is no need
    % for temporal storage
    P([i pos],:) = P([pos i], :);
    if(i ~= pos)
      At([i pos], :) = At([pos i], :);
      L([i pos], 1:i-1) = L([pos i], 1:i-1);
      b([i pos], :) = b([pos i], :);
    end
    while(j<=n)
      L(j,i)=At(j,i)/At(i,i);
      % Combining these rows is easy
      % They are 0 up to column i
      % And combining rows is easy as above
      At(j,i:n) = [0 At(j,i+1:n) - L(j,i)*At(i,i+1:n)];
      bt(j)=bt(j)-bt(i)*L(j,i);
      j=j+1;
    end
    i=i+1;
  end
end
```

LISTING 3.2. *LUP* Factorization

CHAPTER 4

Interpolation

Given a set of data —generally a cloud of points on a plane—, the human *temptation* is to use them as source of knowledge and forecasting. Specifically, given a list of coordinates associated to some kind of event (say, an experiment or a collection of measurements) (x_i, y_i), the "natural" thing to do is to use it for *deducing* or *predicting* the value y would take if x took some other value not in the list. This is the *interpolating and extrapolating tendency* of humans. There is no helping it. The most that can be done is studying the most reasonable ways to perform those forecasts.

We shall use, along this whole chapter, a list of $n+1$ pairs (x_i, y_i)

(6)

x	x_0	x_1	$\dots$	x_{n-1}	x_n
y	y_0	y_1	$\dots$	y_{n-1}	y_n

which is assumed ordered on the **x** coordinates, which are all different, as well: $x_i < x_{i+1}$. The aim is to find functions which somehow have a relation (a kind of *proximity*) to that *cloud of points*.

1. Linear (piecewise) interpolation

The first, simplest and useful idea is to use a piecewise defined function from x_0 to x_n consisting in "joining each point to the next one by a straight line." This is called *piecewise linear interpolation* or *linear spline* —we shall define *spline* in general later on.

DEFINITION 13. The piecewise linear interpolating function for list (6) is the function $f : [x_0, x_n] \to \mathbb{R}$ defined as follows:

$$f(x) = \frac{y_i - y_{i-1}}{x_i - x_{i-1}}(x - x_{i-1}) + y_{i-1} \quad \text{if } x \in [x_{i-1}, x_i]$$

that is, the piecewise defined function whose graph is the union of the linear segments joining (x_{i-1}, y_{i-1}) to (x_i, y_i), for $i = 1, \dots, n$.

Piecewise linear interpolation has a set of properties which make it quite interesting:

- It is *very* easy to compute.
- It passes through all the data points.

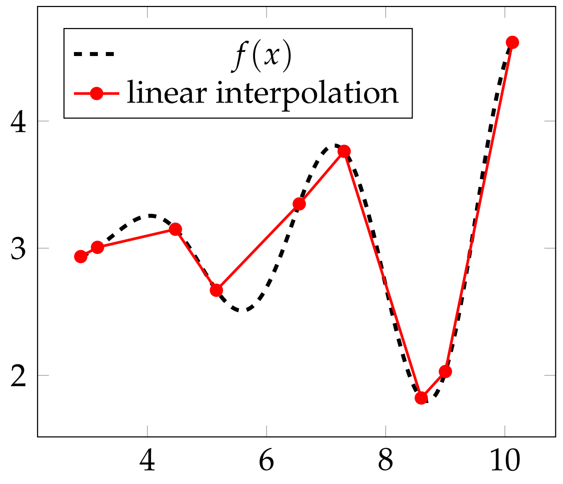

FIGURE 1. Piecewise linear interpolation of a 9-point cloud.

- It is continuous.

That is why it is frequently used for drawing functions (it is what Matlab does by default): if the data cloud is dense, the segments are short and corners will not be noticeable on a plot.

The main drawback of this technique is, precisely, the corners which appear anywhere the cloud of points does not correspond to a straight line. Notice also that this (and the following splines which we shall explain) are *interpolation* methods only, not suitable for *extrapolation*: they are used to approximate values *between the endpoints* x_0, x_n, never outside that interval.

2. Can parabolas be used for this?

Linear interpolation is deemed to give rise to corners whenever the data cloud is not on a straight line. In general, interpolation requirements do not only include the graph to pass through the points in the cloud but also to be *reasonably smooth* (this is not just for aesthetic reasons but also because reality usually behaves this way). One could try and improve linear interpolation with higher degree functions, imposing that the tangents of these functions match at the intersection points. For example, one could try with parabolic segments (polynomials of degree two). As they have three degrees of freedom, one could make them not only pass through (x_{i-1}, y_{i-1}) and (x_i, y_i), but also have the same derivative at x_i. This may seem reasonable but has an inherent undesirable property: it is intrinsically asymmetric (we shall not explain why, the reader should be

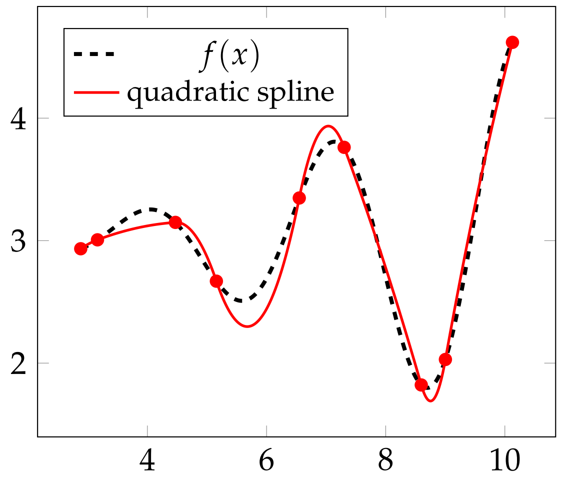

FIGURE 2. Quadratic interpolation of a 9-point cloud. Compare with the original function.

able to realize this after some computations). Without delving into the details, one can verify that this *quadratic spline* is not optimal although it approximates the data cloud without corners.

It has also some other problems (it tends to get far from the data cloud if there are points whose x-coordinates are near but whose y-coordinates differ in different directions). It is almost never used except where the cloud of points is known a priori to resemble a parabola.

The next case, that of *cubic splines* is by far the most used: as a matter of fact, computer aided design (CAD) software uses cubic splines any time it needs to draw a smooth curve, not exactly using a table like (6) but with two tables, because curves are parametrized as $(x(t), y(t))$.

3. Cubic Splines: Continuous Curvature

In order to approximate clouds of points with polynomials of degree 3, the following definition is used:

DEFINITION 14. A *cubic spline* for a table like (6) is a function $f : [x_0, x_n] \to \mathbb{R}$ such that:

- It matches a degree 3 polynomial on every segment $[x_{i-1}, x_i]$, for $i = 1, \dots, n$.
- It is two times differentiable with continuity at every x_i, for $i = 1, \dots, n-1$ (the inner points).

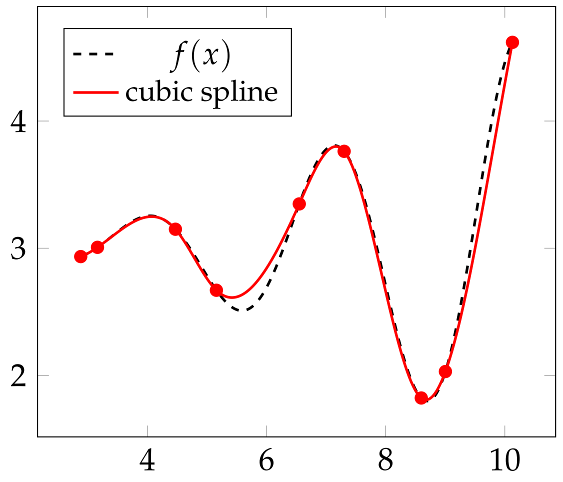

FIGURE 3. Cubic spline for a 9-point cloud. Compare with the original function.

From this follows that, if P_i is the polynomial of degree 3 matching f on $[x_{i-1}, x_i]$, then $P_i'(x_i) = P_{i+1}'(x_i)$ and $P_i''(x_i) = P_{i+1}''(x_i)$; these facts, together with $P_i(x_i) = y_i$ and $P_i(x_{i+1}) = y_{i+1}$ give 4 conditions for each P_i, except for the first and the last ones, for which there are only 3 (this symmetry of behavior at the endpoints is what quadratic splines lack). Thus, the conditions for being a cubic spline *almost* determine the polynomials P_i (and hence f). Another condition for P_1 and P_n is needed for a complete specification. There are several alternatives, some of which are:

- That the second derivative at the endpoints be 0. This gives rise to the *natural cubic spline*, but needs not be the best option for a problem. The corresponding equations are $P_1''(x_0) = 0$ and $P_n''(x_n) = 0$.
- That the *third* derivative matches at the "almost" extremal points: $P_1'''(x_1) = P_2'''(x_1)$ and $P_n'''(x_{n-1}) = P_{n-1}'''(x_{n-1})$. This is the *extrapolated spline*.
- A periodicity condition: $P_1'(x_0) = P_n'(x_n)$ and the same for the second derivative: $P_1''(x_0) = P_n''(x_n)$. This may be relevant if, for example, one is interpolating a periodic function.

3.1. Computing the Cubic Spline: Tridiagonal Matrices. Before proceeding with the explicit computation of the cubic spline, let us introduce some useful notation which will allow us to simplify most of the expressions that will appear.

We shall denote by P_i the polynomial corresponding to the interval $[x_{i-1}, x_i]$ and we shall always write it *relative to* x_{i-1}, as follows:

$$P_i(x) = a_i + b_i(x - x_{i-1}) + c_i(x - x_{i-1})^2 + d_i(x - x_{i-1})^3. \tag{7}$$

We wish to compute a_i, b_i, c_i and d_i for $i = 1, \dots, n$ from the cloud of points $(\mathbf{x}, \mathbf{y})$ above (6).

The next normalization is to rename $x_i - x_{i-1}$ and use

$$h_i = x_i - x_{i-1}, \text{ for } i = 1, \dots, n$$

(that is, use the width of the n intervals instead of the coordinates x_i).

Finally, in an attempt to clarify further, known data will be written in boldface.

We reason as follows:

- The values a_i are directly computable, because $P_i(\mathbf{x_{i-1}}) = a_i$ and, by hypothesis, this must equal $\mathbf{y_{i-1}}$. Hence,

$$a_i = \mathbf{y_{i-1}} \text{ for } i = 1, \dots, n$$

- As $P_i(\mathbf{x_i}) = \mathbf{y_i}$, using now that $a_i \mathbf{y_{i-1}}$ on the right hand side of the equality (7), one gets

$$b_i\mathbf{h_i} + c_i\mathbf{h_i}^2 + d_i\mathbf{h_i}^3 = \mathbf{y_i} - \mathbf{y_{i-1}}. \tag{8}$$

  for $i = 1, \dots, n$, which gives n linear equations.
- The condition on the continuity of the derivative is $P_i'(\mathbf{x_i}) = P_{i+1}'(\mathbf{x_i})$, so that

$$b_i + 2c_i\mathbf{h_i} + 3d_i\mathbf{h_i}^2 = b_{i+1}, \tag{9}$$

  for $i = 1, \dots n-1$. This gives other $n-1$ equations.
- Finally, the second derivatives must match at the intermediate points, so that

$$2c_i + 6d_i\mathbf{h_i} = 2c_{i+1}, \tag{10}$$

  for $i = 1, \dots, n-1$, which gives other $n-1$ equations.

Summing up, there are (apart from the known a_i), $3n-2$ linear equations for $3n$ unknowns (all of the b_i, c_i and d_i). We have already remarked that one usually imposes two conditions at the endpoints in order to get $3n$ linear equations.

However, before including the other conditions, one simplifies and rewrites all the above equations in order to obtain a more *intelligible* system. What is done is to solve the d's and the b's in terms of the c's, obtaining a system in which there are only c's.

First of all, Equations (10) are used to solve the d_i:

$$d_i = \frac{c_{i+1} - c_i}{3\mathbf{h_i}} \tag{11}$$

and substituting in (8), one gets (solving for b_i):

$$b_i = \frac{\mathbf{y_i} - \mathbf{y_{i-1}}}{\mathbf{h_i}} - \mathbf{h_i}\frac{c_{i+1} + 2c_i}{3}; \tag{12}$$

on the other hand, substituting d_i in (8) and computing until solving b_i, one gets

$$b_i = b_{i-1} + \mathbf{h_{i-1}}(c_i + c_{i-1}). \tag{13}$$

for $i = 2, \dots, n$. Now one only needs to use Equations (12) for i and $i-1$ and *introduce* them into (13), so that there are only c's. After some other elementary calculations, one gets

$$\mathbf{h_{i-1}}c_{i-1} + (2\mathbf{h_{i-1}} + 2\mathbf{h_i})c_i + \mathbf{h_i}c_{i+1} = 3\left(\frac{\mathbf{y_i} - \mathbf{y_{i-1}}}{\mathbf{h_i}} - \frac{\mathbf{y_{i-1}} - \mathbf{y_{i-2}}}{\mathbf{h_{i-1}}}\right) \tag{14}$$

for $i = 2, \dots, n-1$.

This is a system of the form $Ac = \alpha$, where A is the $n-2$ by n matrix

$$A = \begin{pmatrix} h_1 & 2(h_1+h_2) & h_2 & 0 & \dots & 0 & 0 & 0 \\ 0 & h_2 & 2(h_2+h_3) & h_3 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & h_{n-1} & 2(h_{n-1}+h_n) & h_n \end{pmatrix} \tag{15}$$

and c is the column vector $(c_1, \dots, c_n)^t$, whereas α is

$$\begin{pmatrix} \alpha_2 \\ \alpha_3 \\ \vdots \\ \alpha_{n-1} \end{pmatrix}$$

with

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

It is easy to see that this is a consistent system (with infinite solutions, because there are two missing equations). The equations which are usually added were explained above. If one sets, for example, $c_1 = 0$ and $c_n = 0$, so that A is completed above with a row $(1\ 0\ \dots\ 0)$and below with $(0\ 0\ 0\ \dots\ 1)$, whereas α gets a top and bottom 0. From these n equations, one computes all the c_i and, using (11) and (12), one gets all the b's and d's.

The above system (15), which has only nonzero elements on the diagonal and the two adjacent lines is called *tridiagonal*. These systems are easily solved using *LU* factorization and one can even compute the solution directly, solving the c's in terms of the α's and h's. One might as well use iterative methods but for these very simple systems, *LU* factorization is fast enough.

3.2. The Algorithm. We can now state the algorithm for computing the interpolating cubic spline for a data list $\mathbf{x}, \mathbf{y}$ of length $n+1$, $\mathbf{x} = (x_0, \dots, x_n)$, $\mathbf{y} = (y_0, \dots, y_n)$, in which $x_i < x_{i+1}$ (so that all the values in $\mathbf{x}$ are different). This is Algorithm 9.

3.3. Bounding the Error. The fact that the cubic spline is graphically *satisfactory* does not mean that it is *technically useful*. As a matter of fact, it is much more useful than what it might seem. If a function is "well behaved" on the fourth derivative, then the cubic spline is a very good approximation to it (and as the intervals get smaller, the better the approximation is). Specifically, *for clamped cubic splines*, we have:

THEOREM 6. *Let $f : [a,b] \to \mathbb{R}$ be a 4 times differentiable function with $|f^{4)}(x)| < M$ for $x \in [a,b]$. Let h be the maximum of $x_i - x_{i-1}$ for $i = 1, \dots, n$. If $s(x)$ is a the clamped cubic spline for $(x_i, f(x_i))$, then*

$$|s(x) - f(x)| \leq \frac{5M}{384}h^4.$$

This result can be most useful for computing integrals and bounding the error or for bounding the error when interpolating values of solutions of differential equations. Notice that the clamped cubic spline for a function f is such that $s'(x_0) = f'(x_0)$ and $s'(x_n) = f'(x_n)$, that is, the first derivative at the endpoints is given by the first derivative of the interpolated function.

Notice that this implies, for example, that if $h = 0.1$ and $M < 60$ (which is rather common: a derivative greater than 60 is huge), then the distance *at any point* between the original function f and the interpolating cubic spline is less than 10^{-4}.

3.4. General Definition of Spline. We promised to give the general definition of spline:

DEFINITION 15. Given a data list as (6), an *interpolating spline of degree m* for it, (for $m > 0$), is a function $f : [x_0, x_n] \to \mathbb{R}$ such that

Algorithm 9 Computation of the Cubic Spline

```
  Input: a data table $\mathbf{x}, \mathbf{y}$ as specified in (6) and two conditions, one on the first and one on the last node, usually.
  Output: an interpolating cubic spline for the table. Specifically, a $n$ lists of four numbers $(a_i, b_i, c_i, d_i)$ such that the polynomials $P_i(x) = a_i + b_i(x - x_{i-1}) + c_i(x - x_{i-1})^2 + d_i(x - x_{i-1})^3$ give an interpolating cubic spline for the table, satisfying the two extra conditions of the input.
⋆START
  $a_i \leftarrow y_{i-1}$ for $i$ from 1 to $n$
  $h_i \leftarrow x_i - x_{i-1}$ for $i$ from 1 to $n$
  $i \leftarrow 1$
  [Build the tridiagonal system:]
  while $i \leq n$ do
      if $i > 1$ and $i < n$ then
          $F_i \leftarrow (0 \cdots 0\ h_{i-1}\ 2(h_{i-1} + h_i)\ h_i\ 0 \cdots 0)$
          $\alpha_i = 3(y_i - y_{i-1})/h_i - 3(y_{i-1} - y_{i-2})/h_{i-1}$
      else
          $F_i \leftarrow$ the row corresponding to the equation for $P_i$
          $\alpha_i$ the coefficient corresponding to the equation for $P_i$
      end if
      $i \leftarrow i + 1$
  end while
  $M \leftarrow$ the matrix whose rows are $F_i$ for $i = 1$ to $n$
  $c \leftarrow M^{-1}\alpha$ [solve the system $Mc = \alpha$]
  $i \leftarrow 1$
  [Solve the $b'$s and $d'$s:]
  while $i < n$ do
      $b_i \leftarrow (y_i - y_{i-1})/h_i - h_i(c_{i+1} + 2c_i)/3$
      $d_i \leftarrow (c_{i+1} - c_i)/(3h_i)$
      $i \leftarrow i + 1$
  end while
  $b_n \leftarrow b_{n-1} + h_{n-1}(c_n + c_{n-1})$
  $d_n \leftarrow (y_n - y_{n-1} - b_n h_n - c_n h_n^2)/h_n^3$
  return $(a, b, c, d)$
```

- It passes through all the points: $f(x_i) = y_i$,
- Is $m - 1$ times differentiable
- On every interval $[x_i, x_{i+1}]$ it coincides with a polynomial of degree at most m.

That is, a piecewise polynomial function passing through all the points and $m-1$ times differentiable (where 0 times differentiable means continuity).

The most used are linear splines (degree 1) and cubic splines (degree 3). Notice that there is *not* a unique spline of degree m for $m \geq 2$.

4. The Lagrange Interpolating Polynomial

Splines, as has been explained, are *piecewise* polynomial functions which pass through a given set of points, with some differentiability conditions.

One could state a more stringent problem: finding a true polynomial of the least degree which passes through each of the points given by (6). That is, given $n+1$ points, find a polynomial of degree at most n which passes through each and every point. This is the *Lagrange interpolation problem*, which has a solution:

THEOREM 7 (Lagrange's interpolation polynomial). *Given a data list as (6) (recall that $x_i < x_{i+1}$), there is a unique polynomial of degree at most n passing through each (x_i, y_i) for $i = 0, \dots, n$.*

The proof is elementary. First of all, one starts with a simpler version of the problem: given $n+1$ values $x_0 < \dots < x_n$, is there a polynomial of degree at most n whose value is 0 at every x_j for $j = 0, \dots, n$ but for x_i where its value is 1? This is easy because there is a simple polynomial of degree n whose value is 0 at each x_j for $j \neq i$: namely $\phi_i(x) = (x-x_0)(x-x_1)\dots(x-x_{i-1})(x-x_{i+1})\dots(x-x_n)$. Now one only needs multiply $\phi_i(x)$ by a constant so that its value at x_i is 1. The polynomial $\phi_i(x)$ has, at x_i the value

$$\phi_i(x_i) = (x_i - x_1)\dots(x_i - x_{i-1})(x_i - x_{i+1})\dots(x_i - x_n) = \prod_{j\neq i}(x_i - x_j),$$

so that the following polynomial $p_{i(x)}$ takes the value 1 at x_i and 0 at any x_j for $j \neq i$.

$$p_i(x) = \frac{\prod_{j\neq i}(x - x_j)}{\prod_{j\neq i}(x_i - x_j)}. \tag{16}$$

These polynomials $p_0(x), \dots, p_n(x)$ are called the *Lagrange basis polynomials* (there are $n+1$ of them, one for each $i = 0, \dots, n$). The collection $\{p_0(x), \dots, p_n(x)\}$ can be viewed as the basis of the vector space $\mathbb{R}^{n+1}$. From this point of view, now the vector $P(x) = (y_0, y_1, \dots, y_n)$

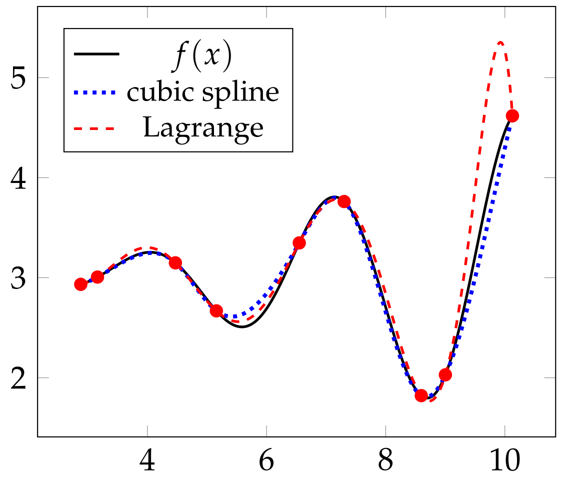

FIGURE 4. Comparison between a cubic spline and the Lagrange interpolating polynomial for the same $f(x)$ as above. Notice the relatively large spike of the Lagrange polynomial at the last interval.

is to be expressed as a linear combination of them, but there is only one way to do so:

$$\begin{aligned} P(x) &= y_0p_0(x) + y_1p_1(x) + \dots + y_np_n(x) = \sum_{i=0}^{n} y_ip_i(x) \\ &= \sum_{i=0}^{n} y_i \frac{\prod_{j\neq i}(x - x_j)}{\prod_{j\neq i}(x_i - x_j)}. \end{aligned} \tag{17}$$

One verifies easily that this $P(x)$ passes through all the points (x_i, y_i) for $i = 0, \dots, n$.

The fact that there is only one polynomial of the same degree as $P(x)$ satisfying that condition can proved as follows: if there existed $Q(x)$ of degree less than or equal to n passing through all those points, the difference $P(x) - Q(x)$ between them would be a polynomial of degree at most n with $n + 1$ zeros and hence, would be 0. So, $P(x) - Q(x)$ would be 0, which implies that $Q(x)$ equals $P(x)$.

Compare the cubic spline interpolation with the Lagrange interpolating polynomial for the same function as before in figure 4.

The main drawbacks of Lagrange's interpolating polynomial are:

- There may appear very small denominators, which may give rise to (large) rounding errors.
- It is too *twisted*.

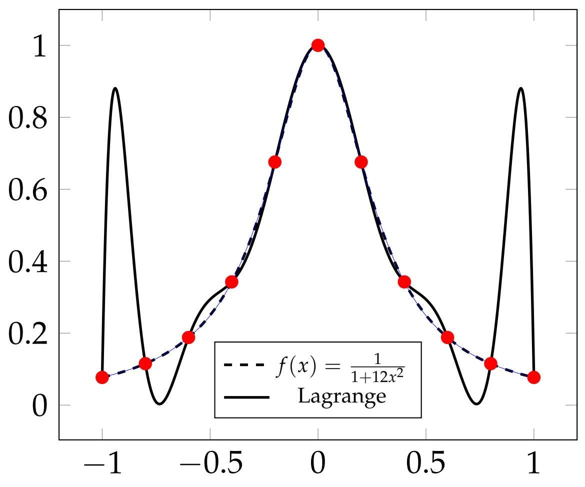

FIGURE 5. Runge's Phenomenon: the Lagrange interpolating polynomial takes values very far away from the original function if the nodes are evenly distributed. There is a thin blue line which corresponds to the cubic spline, which is indistinguishable from the original function.

The first problem is intrinsic to the way we have computed it[1]. The second one depends on the distribution of the x_i in the segment $[x_0, x_n]$. A remarkable example is given by *Runge's phenomenon*: whenever a function with large derivatives is approximated by a Lagrange interpolating polynomial with the $x-$coordinates evenly distributed, then this polynomial gets values which differ greatly from those of the original function (even though it passes through the interpolation points). This phenomenon is much less frequent in splines.

One may try to avoid this issue (which is essentially one of curvature) using techniques which try to minimize the maximum value taken by the polynomial

$$R(x) = (x - x_0)(x - x_1) \dots (x - x_n),$$

that is, looking for a distribution of the x_i which solves a *minimax* problem. We shall not get into details. Essentially, one has:

[1]There are different ways, but we shall not describe them.

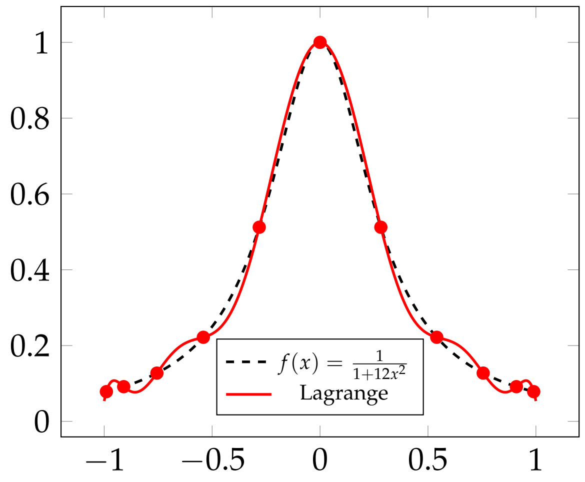

FIGURE 6. Approximation of Runge's function using the Lagrange polynomial and Chebychev's nodes. The maximum error is much less than in 5.

LEMMA 8. *The points $x_0, \dots, x_n$ minimizing the largest maximum value of $R(x)$ on the interval $[-1,1]$ are given by the formula*

$$x_i = \cos\left(\frac{2k+1}{n+1}\frac{\pi}{2}\right)$$

So that, the corresponding points for $[a,b]$, for $a, b \in \mathbb{R}$, are

$$\tilde{x}_i = \frac{(b-a)x_i + (a+b)}{2}.$$

The points x_i in the lemma are called *Chebychev's nodes* for the interval $[a,b]$. They are the ones to be used if one wishes to interpolate a function using Lagrange's polynomial and get a good approximation.

5. Approximate Interpolation

In experimental and statistical problems, data are always considered inexact, so that trying to find a curve fitting them all exactly makes no sense at all. This gives rise to a different interpolation concept: one is no longer interested in making a curve pass through each point in a cloud but in studying what curve in a family *resembles* that cloud in the best way. In our case, when one has a data table like (6), this "resemblance" is measured using the minimal quadratic distance from $f(x_i)$ to y_i, where f is the interpolating function, but this is not the only way to measure the "nearness" (however, it is the one we shall use).

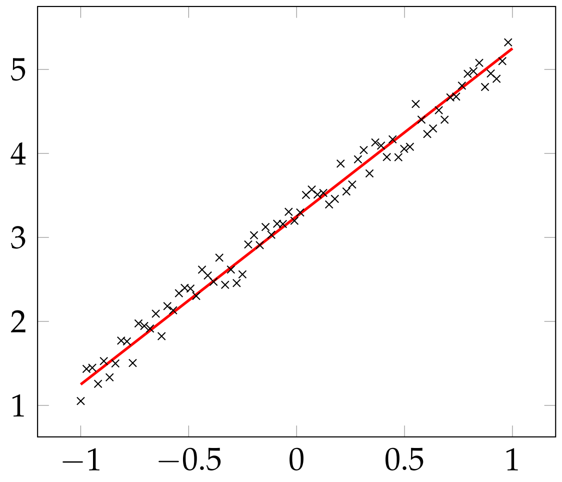

FIGURE 7. Least squares interpolation of a cloud of points using functions of the form $y = ax + b$.

5.1. Least Squares Interpolation. The most common approximate interpolating method is *least squares interpolation*. One starts with a cloud of points $\mathbf{x}, \mathbf{y}$, where $\mathbf{x}$ e $\mathbf{y}$ are arbitrary vectors of length N. This is different from (6): now we enumerate x_i and y_i from 1 to N (notice also that there may be repeated $x-$values and they need not be sorted). Given $f : \mathbb{R} \to \mathbb{R}$, one defines

DEFINITION. The *quadratic error* of f at x_i (one of the coordinates of $\mathbf{x}$)) is the number $(f(x_i) - y_i)^2$. The *total quadratic error* of f on the cloud $(\mathbf{x}, \mathbf{y})$ is the sum

$$\sum_{i=1}^{N} (f(x_i) - y_i)^2.$$

The *least squares linear interpolation problem* consists in, given the cloud of points (x_i, y_i) and *a family of functions*, to find the function *among those in the family* which minimizes the total quadratic error. This family is assumed to be a finite-dimensional vector space (whence the term *linear* interpolation).

We shall give an example of *nonlinear* interpolation problem, just to show the difficulties inherent to its lack of vector structure.

5.2. Linear Least Squares. Assume one has to approximate a cloud of points of size N with a function f belonging to a vector space V of dimension n and that a basis of V is known: $f_1, \dots, f_n$. We always assume $N > n$ (it is actually much greater, in general).

That is, we try to find a function $f \in V$ such that

$$\sum_{i=1}^{N} (f(x_i) - y_i)^2$$

is minimal. As V is a vector space, this f must be a linear combination of the elements of the basis:

$$f = a_1 f_1 + a_2 f_2 + \dots + a_n f_n.$$

And the problem consists in finding the coefficients $a_1, \dots, a_n$ minimizing the value of the $n-$variable function

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

which is an $n-$dimensional optimization problem. It is obvious that F is differentiable function, so that the minimum will annul all the partial derivatives (notice that the variables are the a_i!). The following system, then, needs to be solved (the critical values of F):

$$\frac{\partial F}{\partial a_1} = 0, \frac{\partial F}{\partial a_2} = 0, \dots, \frac{\partial F}{\partial a_n} = 0.$$

The expansion of the partial derivative of F with respect to a_j is

$$\frac{\partial F}{\partial a_j} = \sum_{i=1}^{N} 2 f_j(x_i)(a_1 f_1(x_i) + \dots + a_n f_n(x_i) - y_i) = 0.$$

Writing

$$\mathbf{y_j} = \sum_{i=1}^{N} f_j(x_i) y_i,$$

and stating the equations in matrix form, one gets the system

$$\begin{aligned}
&\begin{pmatrix} f_1(x_1) & f_1(x_2) & \dots & f_1(x_N) \\ f_2(x_1) & f_2(x_2) & \dots & f_2(x_N) \\ \vdots & \vdots & \ddots & \vdots \\ f_n(x_1) & f_n(x_2) & \dots & f_n(x_N) \end{pmatrix} \begin{pmatrix} f_1(x_1) & f_2(x_1) & \dots & f_n(x_1) \\ f_1(x_2) & f_2(x_2) & \dots & f_n(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ f_1(x_N) & f_2(x_N) & \dots & f_n(x_N) \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} = \\
&= \begin{pmatrix} f_1(x_1) & f_1(x_2) & \dots & f_1(x_N) \\ f_2(x_1) & f_2(x_2) & \dots & f_2(x_N) \\ \vdots & \vdots & \ddots & \vdots \\ f_n(x_1) & f_n(x_2) & \dots & f_n(x_N) \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{pmatrix} = \begin{pmatrix} \mathbf{y_1} \\ \mathbf{y_2} \\ \vdots \\ \mathbf{y_n} \end{pmatrix},
\end{aligned} \tag{18}$$

which is of the form

$$XX^t a = Xy$$

where X is the matrix whose row i is the list of values of f_i at $x_1, \dots, x_N$ and y is the column vector $(y_1, y_2, \dots, y_N)^t$, that is:

$$X = \begin{pmatrix} f_1(x_1) & f_1(x_2) & \dots & f_1(x_N) \\ f_2(x_1) & f_2(x_2) & \dots & f_2(x_N) \\ \vdots & \vdots & \ddots & \vdots \\ f_n(x_1) & f_n(x_2) & \dots & f_n(x_N) \end{pmatrix}, \; y = \begin{pmatrix} y_1 \\ y_1 \\ \vdots \\ y_N \end{pmatrix}$$

System (18) is consistent system and has a unique solution if there are at least as many points as the dimension of V and the functions f_i generate linearly independent rows (i.e. if the rank of X is n).

However, system (18) will *very likely* be *ill-conditioned*.

5.3. Non-linear Least Squares... Danger. There are many cases in which the interpolating family of functions is not a vector space. A well-known example is the set given by the functions:

$$f(x) = ae^{bx^2}. \tag{19}$$

These are *Gaussian "bells"* for $a, b > 0$. They obviously do not form a vector space and the techniques of last section do not apply.

One could try to "transform" the problem into a linear one, perform the least squares approximation on the transformed problem and "revert" the result. This may be *catastrophic*.

Taking logarithms on both sides of (19), one gets:

$$\log(f(x)) = \log(a) + bx^2 = a' + bx^2,$$

and, if instead of considering functions ae^{bx^2}, one considers $c + dx^2$, and tries to approach the cloud of points $\mathbf{x}, \log(\mathbf{y})$, then one has a classical linear interpolation problem, for which the techniques above apply. If one obtains a solution $a' = \log(a_0)$, $b' = b_0$, then one could think that

$$g(x) = a_0 e^{b_0 x^2}$$

would be a good approximation to the original cloud of points $(\mathbf{x}, \mathbf{y})$. It might be. But it *might not be*. One has to be aware that this approximation may not minimize the total quadratic error.

Notice that, when taking logarithms, $y-$values near 0 are transformed into values near $-\infty$, and these will have a *much greater* importance than they had before the transformation. This is because, when taking logarithms, absolute and relative errors behave in a totally unrelated way. Moreover, if one of $y-$values is 0, there is no way to take logarithms and it would have to be discarded.

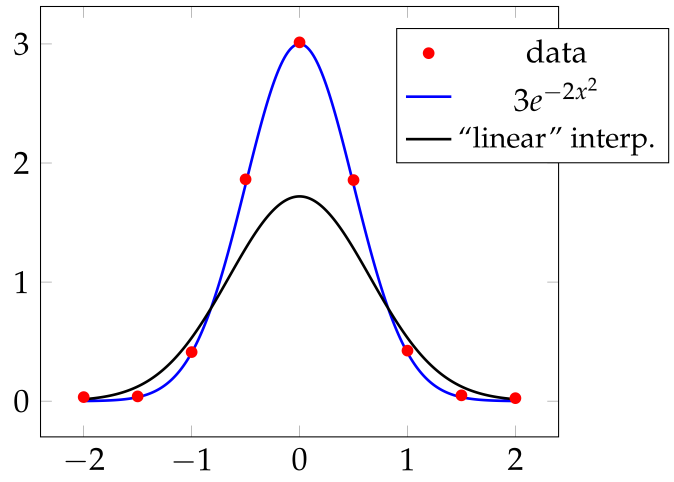

FIGURE 8. Non-linear interpolation using logarithms: the cloud of points resembles the function $f(x)$, but the linear interpolation taking logarithms and performing the inverse transform afterwards (the "low bell") is noticeably bad. This is because there are $y-$values too near to 0.

What is at stake here is, essentially, the fact that $\log(x)$ is not a linear function of x, that is: $\log(a+b) \neq \log(a)+\log(b)$. Then, performing a linear computation on the logarithms and then computing the exponential is not the same as performing the linear operation on the initial data.

6. Code for some of the Algorithms

6.1. Cubic Splines.

Matlab computes *not-a-knot* splines by default (which arise from imposing a condition on the third derivative at the second and last-but-one points) and *natural splines* can only be computed using a *toolbox*. Listing 4.1 implements the computation of natural cubic splines for a cloud of points. Input is:

x: the list of $x-$coordinates
y: the list of $y-$coordinates

It returns a Matlab/Octave "object" which implements a *piecewise polynomial*. There is no need to completely understand this structure, only that if one wants to compute its value at a point, one uses the function `ppval`.

```
% natural cubic spline: second derivative at both
% endpoints is 0. Input is a pair of lists describing
% the cloud of points.
```

```
function [f] = spline_cubico(x, y)
  % safety checks
  n = length(x)-1;
  if(n<=1 | length(x) ~= length(y))
    warning('Wrong data')
    f= [];
    return
  end

  % variables and coefficients for the linear system,
  % these are the ordinary names. Initialization
  a = y(1:n);
  h = diff(x);
  F = zeros(n);
  alpha = zeros(n, 1);

  % true coefficients (and independent terms) of the linear system
  for k=1:n
    if(k> 1& k < n)
      F(k,[k-1 k k+1]) = [h(k-1), 2*(h(k-1) + h(k)), h(k)] ;
      alpha(k) = 3*(y(k+1)-y(k))/h(k) - 3*(y(k) - y(k-1))/h(k-1);
    else
      % these two special cases are the 'natural' condition
      % (second derivatives at endpoints = 0)
      F(k,k) = 1;
      alpha(k) = 0;
    end
    k=k+1;
  end

  % These are the other coefficients of the polynomials
  % (a + bx + cx^2 + dx^3)... Initialization
  c = (F\alpha)';
  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:n-1) = diff(c)./(3*h(1:n-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 4.1. Natural cubic spline computation using Matlab/Octave

The following lines are an example of the usage of Listing 4.1 for approximating the graph of the sine function.

```
> x = linspace(-pi, pi, 10);
> y = sin(x);
> f = spline_cubico(x, y);
> u = linspace(-pi, pi, 400);
> plot(u, ppval(f, u)); hold on;
> plot(u, sin(u), 'r');
```

6.2. The Lagrange Interpolating Polynomial. The computation of the Lagrange Interpolating Polynomial in Matlab/Octave is rather simple, as one can confirm reading listing 4.2.

Input is:

x: The list of $\mathbf{x}-$coordinates of the cloud of points
y: The list of $\mathbf{y}-$coordinates of the cloud of points

Output is a polynomial *in vector form*, that is, a list of the coefficients $a_n, a_{n-1}, \dots, a_0$ such that $P(x) = a_n x^n + \dots + a_1 x + a_0$ is the Lagrange interpolating polynomial for $(\mathbf{x}, \mathbf{y})$.

```
% Lagrange interpolation polynomial
% A single base polynomial is computed at
% each step and then added (multiplied by
% its coefficient) to the final result.
% input is a vector of x coordinates and
% a vector (of equal length) of y coordinates
% output is a polynomial in vector form (help poly).
function [l] = lagrange(x,y)
  n = length(x);
  l = 0;
  for m=1:n
    b = poly(x([1:m-1 m+1:n]));
    c = prod(x(m)-x([1:m-1 m+1:n]));
    l = l + y(m) * b/c;
  end
end
```

LISTING 4.2. Code for computing the Lagrange interpolating polynomial

6.3. Linear Interpolation. For linear interpolation in Matlab/Octave, one has to use a special object, a *cell array*: the list of functions which comprise the basis of the vector space V has to be input *between curly braces*. See Listing 4.3.

Input:

x: The list of $\mathbf{x}-$coordinates of the cloud

y: The list of $\mathbf{y}-$coordinates of the cloud
F: A cell array of anonymous functions. Each elements is one of the functions of the basis of the linear space used for interpolation.

The output is a vector c with coordinates $(c_1, \dots, c_n)$, such that $c_1 F_1 + \cdots + c_n F_n$ is the least squares interpolating function in V for the cloud $(\mathbf{x}, \mathbf{y})$.

```
% interpol.m
% Linear interpolation.
% Given a cloud (x,y), and a Cell Array of functions F,
% return the coefficients of the least squares linear
% interpolation of (x,y) with the base F.
#
% Input:
% x: vector of scalars
% y: vector of scalars
% F: Cell array of anonymous functions
#
% Outuput:
% c: coefficients such that
% c(1)*F{1,1} + c(2)*F{1,2} + ... + c(n)*F{1,n}
% is the LSI function in the linear space <F>.

function [c] = interpol(x, y, F)
  n = length(F);
  m = length(x);
  X = zeros(n, m);
  for k=1:n
    X(k,:) = F{1,k}(x);
  end
  A = X*X.';
  b = X*y.';
  c = (A\b)';
end
```

LISTING 4.3. Code for least squares interpolation

Follows an example of use, interpolating with trigonometric functions

```
> f1=@(x) sin(x); f2=@(x) cos(x); f3=@(x) sin(2*x); f4=@(x) cos(2*x);
> f5=@(x)sin(3*x); f6=@(x)cos(3*x);
> F={f1, f2, f3, f4, f5, f6};
> u=[1,2,3,4,5,6];
> r=@(x) 2*sin(x)+3*cos(x)-4*sin(2*x)-5*cos(3*x);
> v=r(u)+rand(1,6)*.01;
> interpol(u,v,F)
ans =
   1.998522   2.987153  -4.013306  -0.014984  -0.052338  -5.030067
```

CHAPTER 5

Numerical Differentiation and Integration

Numerical differentiation is explained in these notes using the symmetrical increments formula and showing how this can be generalized for higher order derivatives. A brief glimpse into the instability of the problem is given.

Integration is dealt with in deeper detail (because it is easier and much more stable) but also briefly. A simple definition of quadrature formula is given and the easiest ones (the trapezoidal and Simpson's rules) are explained.

1. Numerical Differentiation

Sometimes (for example, when approximating the solution of a differential equation) one has to approximate the value of the derivative of a function at point. A symbolic approach may be unavailable (either because of the software or because of its computational cost) and a numerical recipe may be required. Formulas for approximating the derivative of a function at a point are available (and also for higher order derivatives) but one also needs, in general, bounds for the error incurred. In these notes we shall only show the symmetric rule and explain why the naive approximation is suboptimal.*

instability?

1.1. The Symmetric Formula for the Derivative. The first (simple) idea for approximating the derivative of f at x is to use the definition of derivative as a limit, that is, the following formula:

$$f'(x) \simeq \frac{f(x+h)-f(x)}{h}, \tag{20}$$

where h is a *small increment* of the x variable.

However, the very expression in formula (20) shows its weakness: should one take h positive or negative? This is not irrelevant. Assume $f(x) = 1/x$ and try to compute its derivative at $x = 2$. We shall take $|h| = .01$. Obviously $f'(0.5) = 0.25$. Using the "natural" approximation one has, for $h > 0$:

$$\frac{f(x+.01)-f(x)}{.01} = \frac{\frac{1}{2.01}-\frac{1}{2}}{.01} = -0.248756+$$

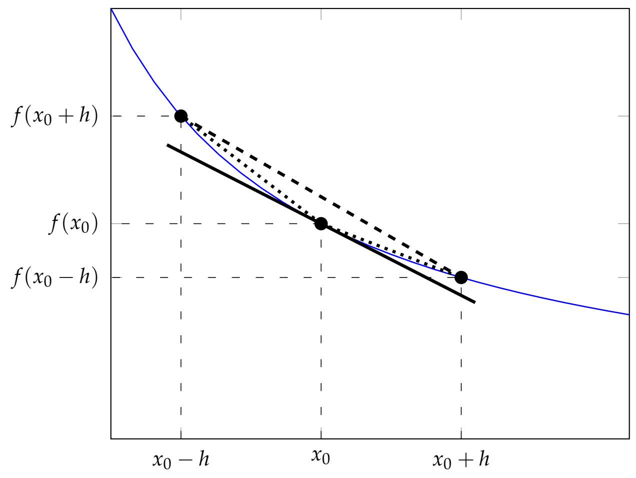

FIGURE 1. Approximate derivative: on the right & left (dotted) and symmetric (dashed). The symmetric one is much similar to the tangent (solid straight line).

whereas, for $h<0$:

$$\frac{f(x-.01)-f(x)}{-.01} = -\frac{\frac{1}{1.99}-\frac{1}{2}}{.01} = -0.251256+$$

which are already different at the second decimal digit. Which of the two is to be preferred? There is no abstract principle leading to choosing one or the other.

Whenever there are two values which approximate a third one and there is no good reason for preferring one to the other, it is reasonable to expect that *the mean value* should be a better approximation than either of the two. In this case:

$$\frac{1}{2}\left(\frac{f(x+h)-f(x)}{h}+\frac{f(x-h)-f(x)}{-h}\right) = -0.2500062+$$

which is an approximation to the real value 0.25 to five significant figures.

This is, actually, the correct way to state the approximation problem to numerical differentiation: to use the symmetric difference around the point and divide by the double of the width of the interval. This is exactly the same as taking the mean value of the "right" and "left" hand derivative.

THEOREM 8. *The naive approximation to the derivative has order* 1 *precision, whereas the symmetric formula has order* 2.

PROOF. Let f be a three times differentiable function on $[x-h, x+h]$. Using the Taylor polynomial of degree 1, one has

$$f(x+h) = f(x) + f'(x)h + \frac{f''(\xi)}{2}h^2,$$

for some ξ in the interval, so that, solving $f'(x)$, one gets

$$f'(x) = \frac{f(x+h)-f(x)}{h} - \frac{f''(\xi)}{2}h,$$

which is exactly the meaning of "having order 1". Notice that the rightmost term cannot be eliminated.

However, for the symmetric formula one can use the Taylor polynomial of degree 2, twice:

$$\begin{aligned} f(x+h) &= f(x) + f'(x)h + \frac{f''(x)}{2}h^2 + \frac{f^{3)}(\xi)}{6}h^3, \\ f(x-h) &= f(x) - f'(x)h + \frac{f''(x)}{2}h^2 - \frac{f^{3)}(\zeta)}{6}h^3 \end{aligned}$$

for some $\xi \in [x-h, x+h]$ and $\zeta \in [x-h, x+h]$. Subtracting:

$$f(x+h) - f(x-h) = 2f'(x)h + K(\xi,\zeta)h^3$$

where K is a sum of the degree 3 coefficients in the previous equation, so that

$$f'(x) = \frac{f(x+h)-f(x-h)}{2h} + \frac{K(\xi,\zeta)}{2}h^2,$$

which is the meaning of "having order 2". □

2. Numerical Integration—Quadrature Formulas

Numerical integration, being a problem in which *errors accumulate* (a "global" problem) is *more stable* than differentiation, surprising as it may seem (even though taking derivatives is much simpler, symbolically, than integration).

In general, one wishes to state an *abstract formula* for performing numerical integration: an algebraic expression such that, given a function $f : [a,b] \to \mathbb{R}$ whose integral is to be computed, one can substitute f in the expression and get a value —the *approximated integral*.

DEFINITION 16. A *simple quadrature formula* for an interval $[a,b]$ is a family of points $x_1 < x_2 < \cdots < x_{n+1} \in [a,b]$ and coefficients (also

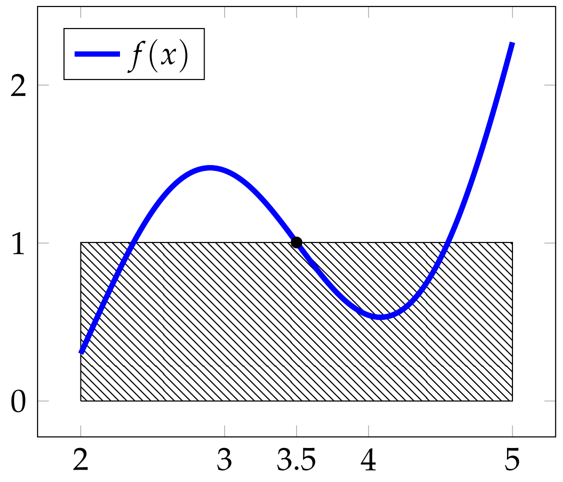

FIGURE 2. Midpoint rule: the area under f is approximated by the rectangle.

called weights) $a_1, \dots, a_{n+1}$. The *approximate integral of a function f on $[a,b]$ using that formula* is the expression

$$a_1 f(x_1) + a_2 f(x_2) + \cdots + a_{n+1} f(x_{n+1}).$$

That is, a quadrature formula is no more than "a way to approximate an integral using intermediate points and weights". The formula is *closed* if $x_1 = a$ and $x_{n+1} = b$; if neither a nor b are part of the intermediate points, then it is *open*.

If the x_i are evenly spaced, then the formula is a *Newton-Coates* formula. These are the only ones we shall explain in this course.

Obviously, one wants the formulas which best approach the integrals of known functions.

DEFINITION 17. A quadrature formula with coefficients $a_1, \dots, a_{n+1}$ is *of order m* if for any polynomial of degree m, one has

$$\int_a^b P(x)\,dx = a_1 P(x_1) + a_2 P(x_2) + \cdots + a_{n+1} P(x_{n+1}).$$

That is, if the formula is exact for *polynomials of degree m*.

The basic quadrature formulas are: the *midpoint* formula (open, $n = 0$), the trapezoidal rule (closed, two points, $n = 1$) and Simpson's formula (closed, three points, $n = 2$).

We first show the *simple* versions and then generalize them to their *composite* versions.

2.1. The Midpoint Rule. A coarse but quite natural way to approximate an integral is to multiply the value of the function at the midpoint by the width of the interval. This is the *midpoint formula*:

DEFINITION 18. The *midpoint quadrature formula* corresponds to $x_1 = (a+b)/2$ and $a_1 = (b-a)$. That is, the approximation

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

given by the area of the rectangle having a horizontal side at $f((b-a)/2)$.

One checks easily that the midpoint rule is of order 1: it is exact for linear polynomials but not for quadratic ones.

2.2. The Trapezoidal Rule. The next natural approximation (which is not necessarily better) is to use two points. As there are two already given (a and b), the naive idea is to use them.

Given a and b, one has the values of f at them. One could interpolate f linearly (using a line) and approximate the value of the integral with the area under the line. Or one could use the mean value between two rectangles (one with height $f(a)$ and the other with height $f(b)$). The fact is that both methods give the same value.

DEFINITION 19. The *trapezoidal rule* for $[a,b]$ corresponds to $x_1 = a$, $x_2 = b$ and weights $a_1 = a_2 = (b-a)/2$. That is, the approximation

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

for the integral of f using the trapeze with a side on $[a,b]$, joining $(a, f(a))$ with $(b, f(b))$ and parallel to OY.

Even though it uses one point more than the midpoint rule, the trapezoidal rule is also of order 1.

2.3. Simpson's Rule. The next natural step involves 3 points instead of 2 and using a parabola instead of a straight line. This method is remarkably precise (it has order 3) and is widely used. It is called *Simpson's Rule*.

DEFINITION 20. *Simpson's rule* is the quadrature formula corresponding to the nodes $x_1 = a, x_2 = (a+b)/2$ and $x_3 = b$, and the weights, corresponding to the correct interpolation of a degree

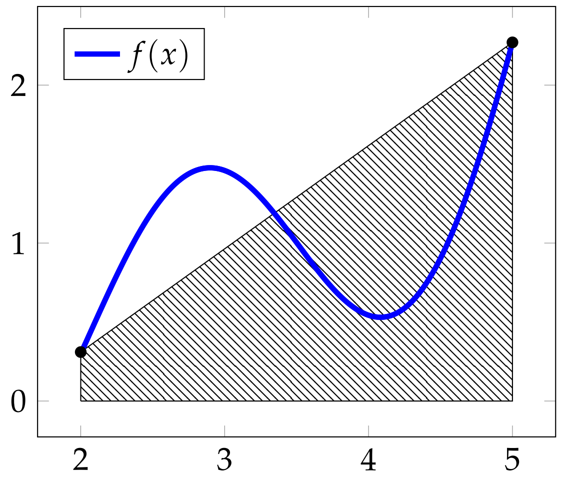

FIGURE 3. trapezoidal rule: approximate the area under f by that of the trapeze.

2 polynomial. That is[1], $a_1 = \frac{b-a}{6}$, $a_2 = \frac{4(b-a)}{6}$ and $a_3 = \frac{b-a}{6}$. Hence, it is the approximation of the integral of f by

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

which is a weighted mean of the areas of three intermediate rectangles. *This rule must be memorized*: one sixth of the length times the values of the function at the endpoints and midpoint with weights $1, 4, 1$.

The remarkable property of Simpson's rule is that it has order 3: even though a parabola is used, the rule integrates correctly polynomials of degree up to 3. Notice that *one does not need to know the equation of the parabola*: the values of f and the weights $(1/6, 4/6, 1/6)$ are enough.

2.4. Composite Formulas. Composite quadrature formulas are no more than "applying the simple ones in subintervals". For example, instead of using the trapezoidal rule for approximating an integral, like

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

one subdivides $[a, b]$ into subintervals and performs the approximation on each of these.

[1]This is an easy computation which the reader should perform by himself.

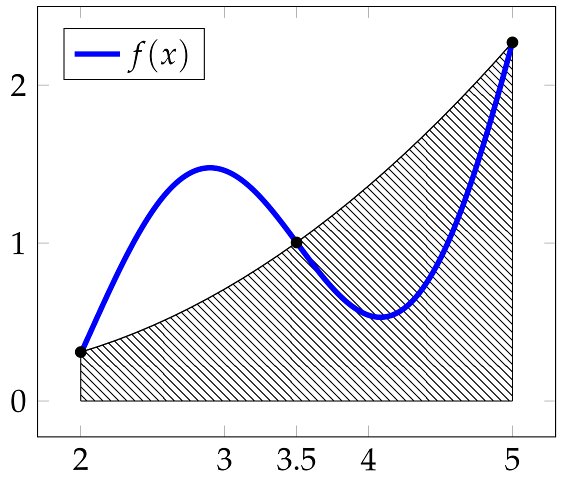

FIGURE 4. Simpson's rule: approximating the area under f by the parabola passing through the endpoints and the midpoint (black).

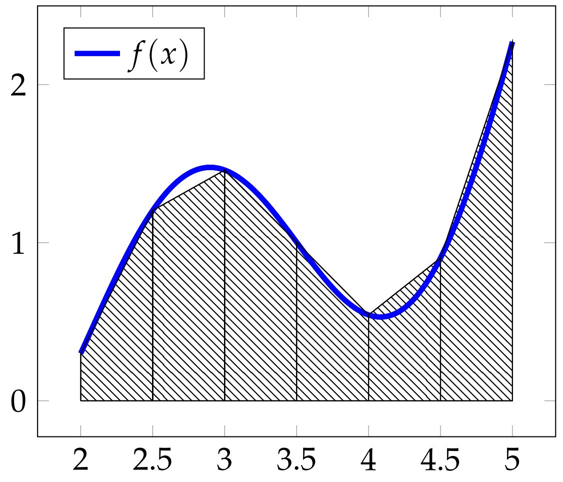

FIGURE 5. Composite trapezoidal rule: just repeat the simple one on each subinterval.

One might do this mechanically, one subinterval after another. The fact is that, due to their additive nature, *it is always simpler to apply the general formula* than to apply the simple one step by step, for closed Newton-Coates formulas (not for open ones, for which the computations are the same one way or the other).

2.4.1. *Composite Trapezoidal Rule.* If there are two consecutive intervals $[a,b]$ and $[b,c]$ *of equal length* (that is, $b-a=c-b$) and the trapezoidal rule is applied on both in order to approximate the integral of f on $[a,c]$, one gets:

$$\int_a^c f(x)\,dx = \frac{b-a}{2}\left(f(b)+f(a)\right) + \frac{c-b}{2}\left(f(c)+f(b)\right) = \frac{h}{2}\left(f(a)+2f(b)+f(c)\right),$$

where $h = b-a$ is the width of each subinterval, either $[a,b]$ or $[b,c]$: the formula just a weighted mean of the values of f at the endpoints and the midpoint. In the general case, when there are more than two subintervals, one gets an analogous formula:

DEFINITION 21 (Composite trapezoidal rule). Given an interval $[a,b]$ and $n+1$ nodes, the *composite trapezoidal rule* for $[a,b]$ with $n+1$ nodes is given by the nodes $x_0 = a, x_1 = a+h, \dots, x_n = b$, the value $h = (b-a)/n$ and the approximation

$$\int_a^c f(x)\,dx \simeq \frac{h}{2}\left(f(a)+2f(x_2)+2f(x_3)+\dots+2f(x_n)+f(b)\right).$$

That is, half the width of the subintervals times the sum of the values at the endpoints and the double of the values at the interior points.

2.4.2. *Composite Simpson's Rule.* In a similar way, the composite Simpson's rule is just the application of Simpson's rule in a sequence of subintervals (which, for the Newton-Coates formula, have all the same width). As before, as the left endpoint of each subinterval is the right endpoint for the next one, the whole composite formula is somewhat simpler to implement than the mere addition of the simple formula on each subinterval.

DEFINITION 22 (Composite Simpson's rule). Given an interval $[a,b]$, divided into n subintervals (so, given $2n+1$ evenly distributed nodes), *Simpson's composite rule* for $[a,b]$ with $2n+1$ nodes is given by the nodes $x_0 = a, x_1 = a+h, \dots, x_{2n} = b$, (where $h = (b-a)/(2n)$) and the approximation

$$\int_a^c f(x)\,dx \simeq \frac{b-a}{6n}\left(f(a)+4f(x_2)+2f(x_3)+4f(x_3)+\dots\right.$$
$$\left.\dots+2f(x_{2n-1})+4f(x_{2n})+f(b)\right).$$

This means that the composite Simpson's rule is the same as the simple version taking into account that one multiplies all by $h/6$, where h is the width of each subinterval and the coefficients are 1 for the two endpoints, 4 for the inner midpoints and 2 for the inner endpoints.

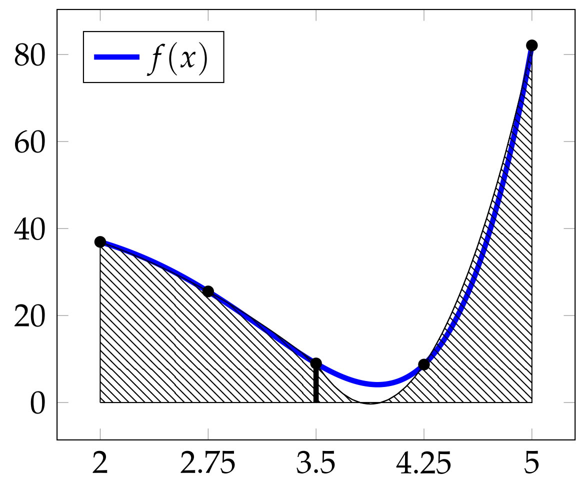

FIGURE 6. Composite Simpson's rule: conceptually, it is just the repetition of the simple rule on each subinterval.

CHAPTER 6

Differential Equations

There is little doubt that the main application of numerical methods is for *integrating differential equations* (which is the technical term for “solving them numerically”).

1. Introduction

A differential equation is a special kind of equation: one in which one of the unknowns is a function. We have already studied some: any integral is a differential equation (it is the simplest kind). For example,

$$y' = x$$

is an equation in which one seeks a function $y(x)$ whose derivative is x. It is well-known that the solution is not unique: there is an *integration constant* and the general solution is written as

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

This integration constant can be better understood graphically. When computing the primitive of a function, one is trying to find a function whose derivative is known. A function can be thought of as a graph on the X, Y plane. The integration constant specifies at which height the graph is. This does not change the derivative, obviously. On the other hand, if one is given the specific problem

$$y' = x, \;\; y(3) = 28,$$

then one is trying to find a function whose derivative is x, *with a condition at a point*: that its value at 3 be 28. Once this value is fixed, *there is only one graph having that shape and passing through* $(3, 28)$. The condition $y(3) = 28$ is called an *initial condition*. One imposes that the graph of f passes through a point and then there is only one f solving the integration problem, which means there is only one suitable constant C. As a matter of fact, C can be computed by substitution:

$$28 = 3^2 + C \;\; \Rightarrow \;\; C = 19.$$

The same idea gives rise to the term *initial condition* for a differential equation.

Consider the equation

$$y' = y$$

(whose general solution should be known). This equation means: find a function $y(x)$ whose derivative is equal to the same function $y(x)$ at every point. One tends to think of the solution as $y(x) = e^x$ but... is this the only possible solution? A geometrical approach may be more useful; the equation means "the function $y(x)$ whose derivative is equal to the height $y(x)$ at each point." From this point of view it seems obvious that there must be more than one solution to the problem: at each point one should be able to draw the corresponding tangent, move a little to the right and do the same. There is nothing special on the points (x, e^x) for them to be the only solution to the problem. Certainly, the general solution to the equation is

$$y(x) = Ce^x,$$

where C is an *integration constant*. If one also specifies an initial condition, say $y(x_0) = y_0$, then necessarily

$$y_0 = Ce^{x_0}$$

so that $C = y_0/e^{x_0}$ is the solution to the *initial value problem* (notice that the denominator is not 0).

This way, in order to find a *unique* solution to a differential equation, one needs to specify *at least* one initial condition. As a matter of fact, the number of these must be the same as the order of the equation.

Consider

$$y'' = -y,$$

whose solutions should also be known: the functions of the form $y(x) = a\sin(x) + b\cos(x)$, for two constants $a, b \in \mathbb{R}$. In order for the solution to be unique, *two* initial conditions must be specified. They are usually stated as the value of y at some point, and the value of the derivative at that same place. For example, $y(0) = 1$ and $y'(0) = -1$. In this case, the solution is $y(x) = -\sin(x) + \cos(x)$.

This chapter deals with *approximate solutions* to differential equations. Specifically, *ordinary differential equations* (i.e. with functions $y(x)$ of a single variable x).

2. The Basics

The first definition is that of differential equation

DEFINITION 23. An *ordinary differential equation* is an equality $A = B$ in which the only unknown is a function of one variable whose derivative of some order appears explicitly.

The adjective *ordinary* is the condition on the unknown of being a function of a single variable (there are no partial derivatives).

EXAMPLE 18. We have shown some examples above. Differential equations can get many forms:

$$\begin{aligned} y' &= \sin(x) \\ xy &= y' - 1 \\ (y')^2 - 2y'' + x^2y &= 0 \\ \frac{y'}{y} - xy &= \cos(y) \end{aligned}$$

etc.

In this chapter, the unknown in the equation will always be denoted with the letter y. The variable on which it depends will usually be either x or t.

DEFINITION 24. A differential equation is of order n if n is the highest order derivative of y appearing in it.

The specific kind of equations we shall study in this chapter are the *solved ones* (which does not mean that they are already solved, but that they are written in a specific way):

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

DEFINITION 25. An *initial value problem* is a differential equation together with an initial condition of the form $y(x_0) = y_0$, where $x_0, y_0 \in \mathbb{R}$.

DEFINITION 26. The *general solution* to a differential equation E is a family of functions $f(x, c)$, where c is one (or several) constants such that:

- Any solution of E has the form $f(x, c)$ for some c.
- Any expression $f(x, c)$ is a solution of E,

except for possibly a finite number of values of c.

If integrating functions of a real variable is already a complex problem, the exact integration a differential equation is, in general, impossible. That is, the explicit computation of the symbolic solution to a differential equation is a problem which is usually not tackled. What one seeks is to know an approximate solution and a *reasonably good* bound for the error incurred when using that approximation instead of the "true" solution.

Notice that, in reality, most of the numbers appearing in the equation describing a problem will already be *inexact*, so trying to get an "exact" solution is already a mistake.

3. Discretization

We shall assume a two-variable function $f(x,y)$ is given, which is defined on a region $x \in [x_0, x_n]$, $y \in [a,b]$, and which satisfies the following condition (which the reader is encouraged to *forget*):

DEFINITION 27. A function $f(x,y)$ defined on a set $X \in \mathbb{R}^2$ satisfies *Lipschitz's condition* if there exists $K > 0$ such that

$$|f(x_1) - f(x_2)| \leq K|x_1 - x_2|$$

for any $x_1, x_2, \in X$, where $|\ |$ denotes the absolute value of a number.

This is a kind of "strong continuity condition" (i.e. it is easier for a function to be continuous than to be Lipschitz). What matters is that this condition has a very important consequence for differential equations: it guarantees the uniqueness of the solution. Let X be a set $[x_0, x_n] \times [a,b]$ (a strip, or a rectangle) and $f(x,y) : X \to \mathbb{R}$ a function on X which satisfies Lipschitz's condition. Then

THEOREM 9 (Cauchy-Kovalevsky). *Under the conditions above, any differential equation $y' = f(x,y)$ with an initial condition $y(x_0) = y_0$ for $y_0 \in (a,b)$ has a unique solution $y = y(x)$ defined on $[x_0, x_0 + t]$ for some $t \in \mathbb{R}$ greater than* 0.

Lipschitz's condition is not so strange. As a matter of fact, polynomials and all the "analytic functions" (exponential, logarithms, trigonometric functions, etc...) and their inverses (where they are defined and continuous) satisfy it. An example which does not is $f(x) = \sqrt{x}$ on an interval containing 0, because f at that point has a "vertical" tangent line. The reader should not worry about this condition (only if he sees a derivative becoming infinity or a point of discontinuity, but we shall not discuss them in these notes). We give just an example:

EXAMPLE 19 (Bourbaki "Functions of a Real Variable", Ch. 4, §1). The differential equation $y' = 2\sqrt{|y|}$ with initial condition $y(0) = 0$ has an infinite number of solutions. For example, any of the following, for $a, b > 0$:

(1) $y(t) = 0$ for any interval $(-b, a)$,
(2) $y(t) = -(t+b)^2$ for $t \leq -b$,
(3) $y(t) = (t-a)^2$ for $t \geq a$

is a solution of that equation. This is because the function on the right hand side, $\sqrt{|y|}$, *is not Lipschitz* near $y = 0$. (The reader is suggested to verify both assertions).

In summary, any "normal" initial value problem has a unique solution. What is difficult is finding this.

And what about an approximation?

3.1. The derivative as an arrow. One is usually told that the derivative of a function of a real variable is *the slope* of its graph at the corresponding point. However, a more useful idea for the present chapter is to think of it as *the Y coordinate of the velocity vector of the graph*.

When plotting a function, one should imagine that one is drawing a curve with constant horizontal speed (because the $OX-$axis is homogeneous, one goes from left to right at uniform speed). This way, the graph of $f(x)$ is actually the plane curve $(x, f(x))$. Its tangent vector at any point is $(1, f'(x))$: the derivative $f'(x)$ of $f(x)$ is the vertical component of this vector.

From this point of view, a differential equation in solved form $y' = f(x, y)$ can be interpreted as the statement "find a curve $(x, y(x))$ such that the velocity vector at each point is $(1, f(x, y))$." One can then draw the family of "velocity" vectors on the plane (x, y) given by $(1, f(x, y))$ (the function $f(x, y)$ is known, remember). This visualization, like in Figure 1, already gives an idea of the shape of the solution.

Given the arrows —the vectors $(1, f(x, y))$— on a plane, drawing a curve whose tangents are those arrows should not be too hard. Even more, if what one needs is just an approximation, instead of drawing a curve, one could draw "little segments going in the direction of the arrows." If these segments have a very small $x-$coordinate, one reckons that an approximation to the solution will be obtained.

This is exactly Euler's idea.

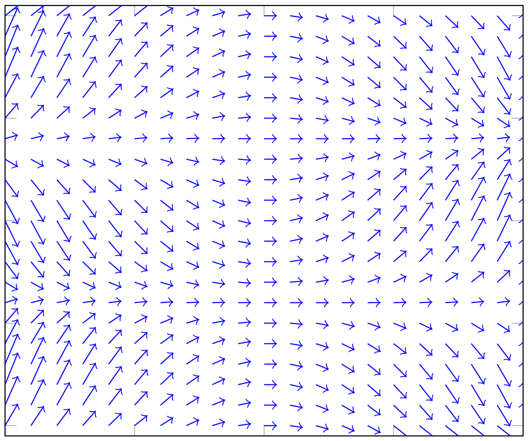

FIGURE 1. Arrows representing vectors of a solved differential equation $y' = f(x,y)$. Notice how the horizontal component is constant while the vertical one is not. Each arrow corresponds to a vector $(1, f(x,y))$ where $f(x,y) \simeq x\cos(y)$.

Given a plane "filled with arrows" indicating the velocity vectors at each point, the simplest idea for drawing the corresponding curve that:

- Passes through a specified point (initial condition $y(x_0) = y_0$)
- Whose $y-$velocity component is $f(x,y)$ at each point...

consists in *discretizing* the $x-$coordinates. Starting at x_0, one assumes that OX is quantized in intervals of width h (constant, by now) "small enough." Now, instead of drawing a *smooth curve*, one approximates it by small steps (of width h) on the $x-$variable.

As the solution has to verify $y(x_0) = y_0$, one needs only

- Draw the point (x_0, y_0) (the initial condition).
- Compute $f(x_0, y_0)$, the *vertical* value of the velocity vector of the solution at the initial point.
- As the $x-$coordinate is quantized, the *next* point of the approximation will have $x-$coordinate equal to $x_0 + h$.
- As the velocity at (x_0, y_0) is $(1, f(x_0, y_0))$, the simplest approximation to the *displacement of the curve in an interval of width h on the $x-$coordinate* is $(h, hf(x_0, y_0))$. Let $x_1 = x_0 + h$ and $y_1 = y_0 + hf(x_0, y_0)$.
- Draw the segment from (x_0, y_0) to (x_1, y_1): this is the first "approximate segment" of the solution.

- At this point one is in the same situation as at the beginning but with (x_1, y_1) instead of (x_0, y_0). Repeat.

And so on as many times as one desires. The above is *Euler's* algorithm for numerical integration of differential equations.

4. Sources of error: truncation and rounding

It is obvious that the solution to an equation $y' = f(x,y)$ will never (or practically so) be a line composed of straight segments. When using Euler's method, there is an intrinsic error which can be analyzed, for example, with Taylor's formula. Assume $f(x,y)$ admits a sufficient number of partial derivatives. Then $y(x)$ will be differentiable also and

$$y(x_0 + h) = y(x_0) + hy'(x_0) + \frac{h^2}{2}y''(\theta)$$

for some $\theta \in [x_0, x_0 + h]$. What Euler's method does is to *remove the last term*, so that the error incurred is exactly that: a term of order 2 in h in the first iteration. This error, which arises from *truncating* the Taylor expansion is called *truncation error* in this setting.

In general, one assumes that the interval along which the integration is performed has width of magnitude h^{-1}. Usually, starting from an interval $[a, b]$ one posits a number of subintervals n (or intermediate points, $n - 1$) and takes $x_0 = a$, $x_n = b$ and $h = (b - a)/n$, so that going from x_0 to x_n requires n iterations and the *global truncation error* has order $h^{-1}O(h^2) \simeq O(h)$.

However, truncation is not the only source of error. Floating-point operations incur always *rounding* errors.

4.1. Details on the truncation and rounding errors.

Specifically, if using IEEE-754, one considers that the smallest *significant* quantity (what is called, the *epsilon*) is $\epsilon = 2^{-52} \simeq 2.2204 \times 10^{-16}$. Thus, the rounding error in each operation is less than ϵ. If "operation" means a step in the Euler algorithm (which may or may not be the case), then each step from x_i to x_{i+1} incurs an error of at most ϵ and hence, the *global rounding error* is bounded by ϵ/h. This is very tricky because it implies that *the smaller the interval h, the larger the rounding error incurred*, which is counterintutive. So, **making h smaller does not necessarily improve the accuracy of the method!**

Summing up, the addition of the truncation and rounding errors can be approximated as

$$E(h) \simeq \frac{\epsilon}{h} + h,$$

which grows both when h becomes larger and when it gets smaller. The minimum $E(h)$ is given by $h \simeq \sqrt{\epsilon}$: which means that there is no point in using intervals of width less than $\sqrt{\epsilon}$ in the Euler method (actually, ever). Taking smaller intervals may perfectly lead to huge errors.

This explanation about truncation and rounding errors is relevant for any method, not just Euler's. One needs to bound both for every method and know how to choose the best h. There are even problems for which a single h is not useful and it has to be modified during the execution of the algorithm. We shall not deal with these problems here (the interested reader should look for the term "stiff differential equation").

5. Quadratures and Integration

Assume that in the differential equation $y' = f(x,y)$, the function f depends only on the variable x. Then, the equation would be written

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

expression which means *y is the function whose derivative is $f(x)$*. That is, the problem is that of computing a primitive. We have already dealt with it in Chapter 5 but it is inherently related to what we are doing for differential equations.

When $f(x,y)$ depends also on y, the relation between the equation and a primitive is harder to perceive but we know (by the Fundamental Theorem of Calculus) that, if $y(x)$ is the solution to the initial value problem $y' = f(x,y)$ with $y(x_0) = y_0$, then, integrating both sides, one gets

$$y(x) = \int_{x_0}^{x} f(t, y(t))\, dt + y_0,$$

which is not a primitive but *looks like it*. In this case, there is no way to approximate the integral using the values of f at intermediate points *because one does not know the value of $y(t)$*. But one can take a similar approach.

6. Euler's Method: Integrate Using the Left Endpoint

One can state Euler's method as in Algorithm 10. If one reads carefully, the gist of each step is to approximate each value y_{i+1} as the previous one y_i plus f evaluated at (x_i, y_i) times the interval width.

That is:

$$\int_{x_i}^{x_{i+1}} f(t,y(t))\,dt \simeq (x_{i+1}-x_i)f(x_i,y_i) = hf(x_i,y_i).$$

If $f(x,y)$ were independent of y, then one would be performing the following approximation (for an interval $[a,b]$):

$$\int_a^b f(t)\,dt = (b-a)f(a),$$

which, for lack of a better name, could be called *the left endpoint rule*: the integral is approximated by the area of the rectangle of height $f(a)$ and width $(b-a)$.

Algorithm 10 Euler's algorithm assuming exact arithmetic.

```
  Input: A function $f(x,y)$, an initial condition $(x_0,y_0)$, an interval
  $[a,b]=[x_0,x_n]$ and a step $h=(x_n-x_0)/n$
  Output: A family of values $y_0,y_1,\dots,y_n$ (which approximate the
  solution to $y'=f(x,y)$ on the net $x_0,\dots,x_n$)
⋆START
  $i \leftarrow 0$
  while $i \leq n$ do
      $y_{i+1} \leftarrow y_i + hf(x_i,y_i)$
      $i \leftarrow i+1$
  end while
  return $(y_0,\dots,y_n)$
```

One might try (as an exercise) to solve the problem "using the right endpoint": this gives rise to what are called the *implicit* methods which we shall not study (but which usually perform better than the *explicit* ones we are going to explain).

7. Modified Euler: the Midpoint Rule

Instead of using the left endpoint of $[x_i,x_{i+1}]$ for integrating and computing y_{i+1}, one might use (and this would be better, as the reader should verify) the midpoint rule somehow. As there is no way to know the intermediate values of $y(t)$ further than x_0, some kind of guesswork has to be done. The method goes as follows:

- Use a point near $P_i=(x_i,y_i)$ whose $x-$coordinate is the midpoint of $[x_i,x_{i+1}]$.
- For lack of a better point, the first approximation is done using Euler's algorithm and one takes a point $Q_i=[x_{i+1},y_i+hf(x_i,y_i)]$.

- Compute the midpoint of the segment $\overline{P_iQ_i}$, which is $(x_i + h/2, y_i + h/2f(x_i, y_i))$. Let k be its $y-$coordinate.
- Use the value of f at that point in order to compute y_{i+1}: this gives the formula $y_{i+1} = y_i + hf(x_i + h/2, k)$.

If, as above, $f(x, y)$ did not depend on y, one verifies easily that the corresponding approximation for the integral is

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

which is exactly the *midpoint rule* of numerical integration.

This method is called *modified Euler's method* and its *order* is 2, the same as Euler's, which implies that the accrued error at x_n is $O(h)$. It is described formally in Algorithm 11.

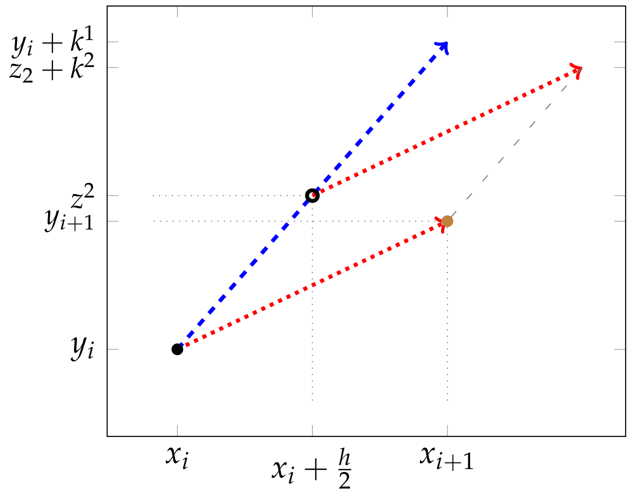

FIGURE 2. Modified Euler's Algorithm. Instead of using the vector $(h, hf(x_i, y_i))$ (dashed), one sums at (x_i, y_i) the dotted vector, which is the tangent vector at the midpoint.

As shown in Figure 2, this method consist in first using Euler's in order to get an initial guess, then computing the value of $f(x, y)$ at the midpoint between (x_0, y_0) and the guess and using this value of f as the approximate slope of the solution at (x_0, y_0). There is still an error but, as the information is gathered "a bit to the right", it is less than the one of Euler's method, analogue to the trapezoidal rule.

The next idea is, instead of using a single vector, compute two of them and use a "mean value:" as a matter of fact, the mean between the vector at the origin and the vector at the end of Euler's method.

Algorithm 11 Modified Euler's algorithm, assuming exact arithmetic.

```
  Input: A function $f(x,y)$, an initial condition $(x_0, y_0)$, an interval
  $[x_0, x_n]$ and a step $h = (x_n - x_0)/n$
  Output: A family of values $y_0, y_1, \dots, y_n$ (which approximate the
  solution of $y' = f(x,y)$ on the net $x_0, \dots, x_n$)
⋆START
  $i \leftarrow 0$
  while $i \leq n$ do
      $k_1 \leftarrow f(x_i, y_i)$
      $z_2 \leftarrow y_i + \frac{h}{2}k_1$
      $k_2 \leftarrow f(x_i + \frac{h}{2}, z_2)$
      $y_{i+1} \leftarrow y_i + hk_2$
      $i \leftarrow i + 1$
  end while
  return $(y_0, \dots, y_n)$
```

8. Heun's Method: the Trapezoidal Rule

Instead of using the midpoint of the segment of Euler's method, one can take the vector corresponding to Euler's method and also the vector at the endpoint of Euler's method and compute the mean of both vectors and use this mean for approximating. This way, one is using the information at the point and some information "later on." This *improves* Euler's method and is called accordingly *improved Euler's method* or *Heun's method*. It is described in Algorithm 12.

At each step one has to perform the following operations:

- Compute $k_1 = f(x_i, y_i)$.
- Compute $z_2 = y_j + hk_1$. This is the coordinate y_{i+1} in Euler's method.
- Compute $k_2 = f(x_{i+1}, z_2)$. This would be the slope at (x_{i+1}, y_{i+1}) with Euler's method.
- Compute the mean of k_1 and k_2: $\frac{k_1+k_2}{2}$ and use this value as "slope". That is, set $y_{i+1} = y_i + \frac{h}{2}(k_1 + k_2)$.

Figure 3 shows a graphical representation of Heun's method.

Figures 4 and 5 show the approximate solutions to the equations $y' = y$ and $y' = -y + \cos(x)$, respectively, using the methods explained in the text, together with the exact solution.

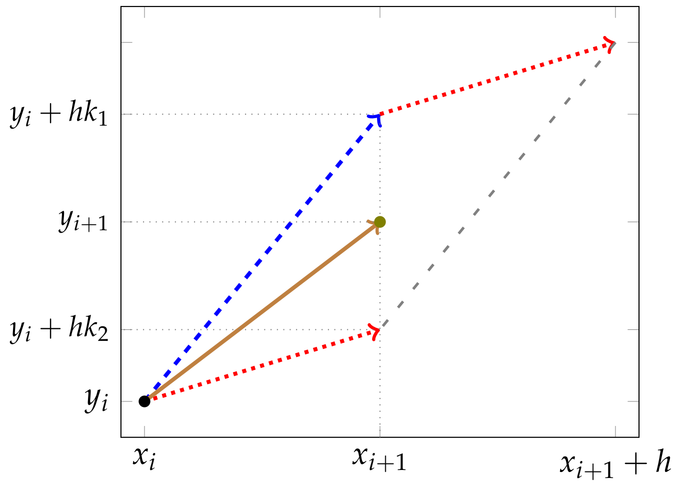

FIGURE 3. Heun's method. At (x_i, y_i) one uses the mean displacement (solid) between the vectors at (x_i, y_i) (dashed) and at the next point $(x_{i+1}, y_i + hk_1)$ in Euler's method (dotted).

Algorithm 12 Heun's algorithm assuming exact arithmetic.

```
  Input: A function $f(x,y)$, an initial condition $(x_0, y_0)$, an interval
  $[x_0, x_n]$ and a step $h = (x_n - x_0)/n$
  Output: A family of points $y_0, y_1, \dots, y_n$( which approximate the
  solution to $y' = f(x,y)$ on the net $x_0, \dots, x_n$)
⋆START
  $i \leftarrow 0$
  while $i \leq n$ do
      $k_1 \leftarrow f(x_i, y_i)$
      $z_2 \leftarrow y_i + hk_1$
      $k_2 \leftarrow f(x_i + h, z_2)$
      $y_{i+1} \leftarrow y_i + \frac{h}{2}(k_1 + k_2)$
      $i \leftarrow i + 1$
  end while
  return $(y_0, \dots, y_n)$
```

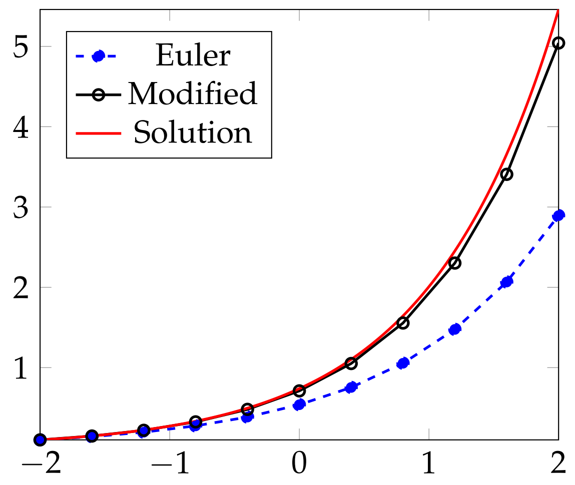

FIGURE 4. Comparison between Euler's and Modified Euler's methods and the true solution to the ODE $y' = y$ for $y(-2) = 10^{-1}$.

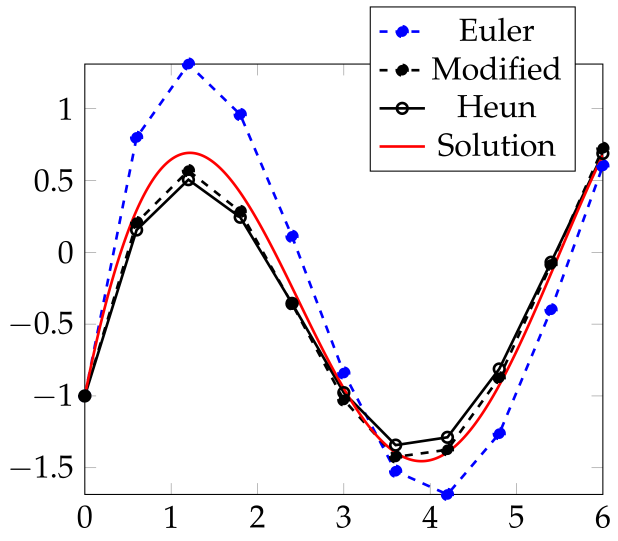

FIGURE 5. Comparison between Euler's, Modified Euler's and Heun's methods and the true solution to the ODE $y' = -y + \cos(x)$ for $y(0) = -1$.

CHAPTER 7

Multivariate and higher order ODEs

In Chapter 6 we dealt with the simplest case of Ordinary Differential Equations: those of a single dependent variable. As a matter of fact, in most cases, the number of dependent variables is more than one and, as we shall see later, any ODE of order greater than one can be transformed into a system of equations of order one with more than one dependent variable. This chapter gives a summary introduction to the numerical solution of these problems. The expectation is that the student gets acquainted with problems of order greater than one and understands their transformation into order one equations.

1. A two-variable example

Let us consider a problem in two variables. Usually —and we shall follow the convention in this chapter— the independent variable is called t, reflecting the common trait of ODEs of being equations which define a motion in terms of time.

Consider a dimensionless body B —a mass point— moving with velocity v in the plane. Hence, if the coordinates of B at time t are $B(t) = (x(t), y(t))$, we shall denote $v = (\dot{x}(t), \dot{y}(t))$. Assume we know, for whatever reason, that the velocity vector satisfies some condition F which depends on the position of B in the plane. This can be expressed as $v = F(x, y)$. The function F is then a vector function of two variables, $F(x, y) = (F_1(x, y), F_2(x, y))$, and we can write the condition on v as:

$$\left\{ \begin{array}{l} \dot{x}(t) = F_1(x(t), y(t)) \\ \dot{y}(t) = F_2(x(t), y(t)) \end{array} \right. \tag{21}$$

which means, exactly, that the $x-$component of the velocity depends on the position at time t as the value of F_1 at the point and that the $y-$component depends as the value of F_2. If we want, writing $B(t) = (x(t), y(t))$ as above, expression 21 can be written, in a compact way as

$$v(t) = F(B(t))$$

which reflects the idea that *what we are doing is just the same as in Chapter 6, only with more coordinates*. This is something that has to be clear from the beginning: the only added difficulty is the number of computations to be carried out.

Notice that in the example above, F depends only on x and y —not on t. However, it might as well depend on t (because the behaviour of the system may depend on time), so that in general, we should write

$$v(t) = F(t, B(t)).$$

Just for completeness, if F does not depend on t, the system is called *autonomous*, whereas if it does depend on t, it is called *autonomous*.

1.1. A specific example. Consider, to be more precise, the same autonomous problem as above but with $F(x,y) = (-y,x)$. This gives the following equations

$$\left\{ \begin{array}{l} \dot{x}(t) = -y(t) \\ \dot{y}(t) = x(t). \end{array} \right. \tag{22}$$

It is clear that, in order to have an initial value problem, we need an initial condition. Because the equation is of order one, we need just the starting point of the motion, that is: two cordinates. Let us take $x(0) = 1$ and $y(0) = 0$ as the initial position for time $t = 0$. The initial value problem is, then

$$\left\{ \begin{array}{l} \dot{x}(t) = -y(t) \\ \dot{y}(t) = x(t). \end{array} \right. \quad \text{with } (x(0), y(0)) = (1, 0) \tag{23}$$

which, after some scaling (so that the plot looks more or less nice) describes the vector field depicted in Figure 1. It may be easy to guess that a body whose velocity is described by the arrows in the diagram follows a circular motion. This is what we are going to prove, actually.

The initial value problem (23) means, from a symbolic point of view, that the functions $x(t)$ and $y(t)$ satisfy the following conditions:

- First, the derivative of $x(t)$ with respect to t is $-y(t)$.
- Then, the derivative of $y(t)$ with respect to t is $x(t)$.
- And finally, the initial values are 1 and 0, for $t = 0$, respectively.

The first two conditions imply that $\ddot{x}(t) = -x(t)$ and $\ddot{y}(t) = -y(t)$. This leads one to think of trigonometric functions and, actually, it is

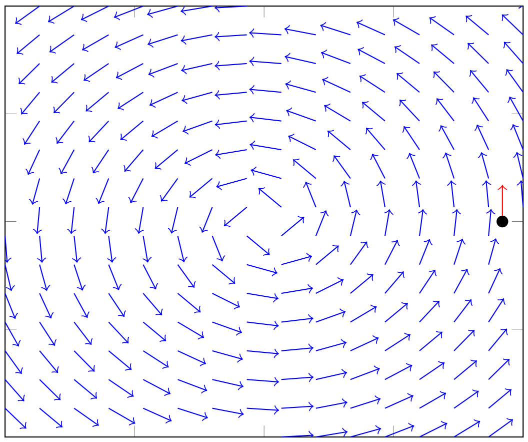

FIGURE 1. Vector field representing (after scaling) the ODE of Equation (23). The initial condition (with the vector at that point) is marked with a black dot.

easy to check that $x(t) = \cos(t)$, $y(t) = \sin(t)$ verify the conditions above. That is, the solution to the initial value problem (23) is

$$\left\{ \begin{aligned} x(t) &= \cos(t) \\ y(t) &= \sin(t) \end{aligned} \right. \tag{24}$$

which, as the reader will have alredy realized, is a circular trajectory around the origin, passing through $(1, 0)$ at time $t = 0$. Any other initial condition $(x(0), y(0)) = (a, b)$ gives rise to a circular trajectory starting at (a, b).

However, our purpose, as we have stated repeatedly in these notes, is not to find a *symbolic* solution to any problem, but to approximate it using the tools at hand. In this specific case, which is of dimension two, one can easily describe the generalization of Euler's method of Chapter 6 to the problem under study. Let us fix a discretization of t, say in steps of size h. Then, a rough approximation to a solution would be:

(1) We start with $x_0 = 1, y_0 = 0$.
(2) At that point, the differential equation means that the velocity $v(t) = (\dot{x}(t), \dot{y}(t))$ is
$$\dot{x}(0) = 0,\ \dot{y}(0) = 1.$$
(3) Because t moves in steps of size h, the trajectory $(x(t), y(t))$ can only be approximated by moving as much as the vector at $t = 0$ says multiplied by the timespan h, hence the next

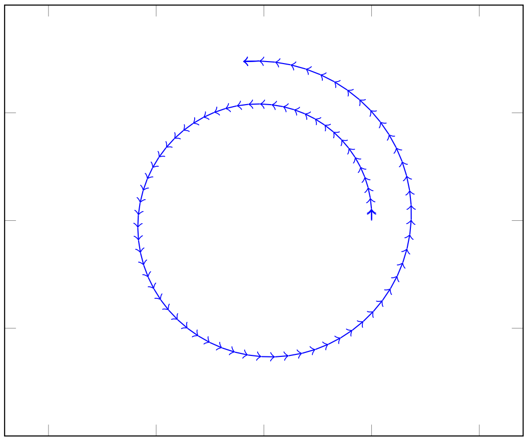

FIGURE 2. Approximation to the solution of Problem (23) using Euler's method. Notice the (obvious) error.

approximate position, (x_1, y_1) is given by

$$(x_1, y_1) = (x_0, y_0) + h \cdot (\dot{x}(0), \dot{y}(0)) = (1, 0) + (h \cdot 0, h \cdot 1) = (0, h).$$

(4) Now, at the point $(x_1, y_1) = (0, h)$, we carry out the analogous computations, taking into account that at this point, $v = (-\cos(h), \sin(0)) \dots$

If we follow the steps just described, we get a picture like Figure 2, in which the step is $h = 0.1$ and the error incurred is easily noticed.

The fact that this method, for this example, gives what looks like a bad approximation needs not make one reject it straightway. There is a clear reason for this, which we shall not delve into but which has to deal with the solutions being always convex —i.e. the trajectories describing the true solutions are always curved in the same way. This makes the Euler method accumulate the errors always, instead of having some positive errors and some negative ones.

2. Multivariate equations: Euler and Heun's methods

The example with two coordinates above is easily generalized to n coordinates. Consider an initial value problem

$$\begin{cases} \dot{x}_1 = f_1(t, x_1, \dots, x_n), & x_1(0) = a_1 \\ \dot{x}_2 = f_2(t, x_1, \dots, x_n), & x_2(0) = a_2 \\ \vdots \\ \dot{x}_n = f_n(t, x_1, \dots, x_n), & x_n(0) = a_n \end{cases} \tag{25}$$

where $f_1(t, x_1, \dots, x_n), \dots, f_n(t, x_1, \dots, x_n)$ are continuous functions of $n+1$ variables —"time", so to say, and "position". We do not assume the system is autonomous. Our aim is to compute an approximate solution using increments of the independent variable t of size h. The generalization of Euler's method is just a copy of what was described above:

(1) Start with $i = 0$ and let $x_{0,1} = a_1, \dots, x_{0,n} = a_n$. Notice that the first subindex indicates the number of the iteration and the second is the coordinate.
(2) Set $t_i = h \cdot i$.
(3) Compute the coordinates of the derivative: that is, compute $v_1 = f_1(t_i, x_{i,1}(t_i), \dots, x_{i,n}(t_i)), \dots, v_n = f_n(t_i, x_{i,1}(t_i), \dots, x_{i,n}(t_i))$. This is much simpler than it seems.
(4) Compute the next point: $x_{i+1,1} = x_{i,1} + h \cdot v_1, x_{i,2} = x_{i,2} + h \cdot v_2, \dots, x_{i+1,n} = x_{i,n} + h \cdot v_n$.
(5) Increase $i = i + 1$ and goto step 2 until one stops.

These steps are formally stated in Algorithm 13.

Algorithm 13 Euler's algorithm for n coordinates. Notice that boldface elements denote vectors.

```
  **Input:** $n$ functions of $n+1$ variables: $f_1(t, x_1, \dots, x_n), \dots, f_n(t, x_1, \dots, x_n)$, an initial condition $(a_1, a_2, \dots, a_n)$, an interval $[A, B]$ and a step $h = (B - A)/N$
  **Output:** A family of vector values $\mathbf{x}_0, \dots, \mathbf{x_N}$ (which approximate the solution to the corresponding initial value problem for $t \in [A, B]$)
⋆START
  $i \leftarrow 0, t_0 \leftarrow A, \mathbf{x}_0 \leftarrow (a_1, \dots, a_n)$
  while $i < N$ do
      $\mathbf{v} \leftarrow (f_1(t_i, \mathbf{x}_i), \dots, f_n(t_i, \mathbf{x}_i))$
      $\mathbf{x}_{i+1} \leftarrow \mathbf{x}_i + h \cdot \mathbf{v}_i$
      $i \leftarrow i + 1$
  end while
  return $\mathbf{x}_0, \dots, \mathbf{x}_N$
```

As we did in Chapter 6, we could state a *modified* version of Euler's method but we prefer to go straightway for Heun's. It is precisely stated in Algorithm 14.

Algorithm 14 Heun's algorithm for n coordinates. Notice that boldface elements denote vectors.

```
  Input: $n$ functions of $n$ variables: $f_1(t, x_1, \dots, x_n), \dots, f_n(t, x_1, \dots, x_n)$, an initial condition $(a_1, a_2, \dots, a_n)$, an interval $[A, B]$ and a step $h = (B - A)/N$
  Output: A family of vector values $\mathbf{x}_0, \dots, \mathbf{x_N}$ (which approximate the solution to the corresponding initial value problem for $t \in [A, B]$)
⋆START
  $i \leftarrow 0, t_0 \leftarrow A, \mathbf{x}_0 \leftarrow (a_1, \dots, a_n)$
  while $i < N$ do
      $\mathbf{v} \leftarrow (f_1(t_i, \mathbf{x}_i), \dots, f_n(t_i, \mathbf{x}_i))$
      $\mathbf{z} \leftarrow \mathbf{x}_i + h \cdot \mathbf{v}_i$
      $\mathbf{w} \leftarrow (f_1(t_i, \mathbf{z}_i), \dots, f_n(t_i, \mathbf{z}_i))$
      $\mathbf{x}_{i+1} \leftarrow \mathbf{x}_i + \frac{h}{2} \cdot (\mathbf{v} + \mathbf{w})$
      $i \leftarrow i + 1$
  end while
  return $\mathbf{x}_0, \dots, \mathbf{x}_N$
```

In Figure 3 a plot of the approximate solution to the initial value problem of Equation 23 is shown. Notice that the approximate trajectory is —on the printed page— indistinguishable from a true circle (we have plotted more than a single loop to show how the approximate solution overlaps to the naked eye). This shows the improved accuracy of Heun's algorithm —although it does not behave so nicely in the general case.

3. From greater order to order one

Ordinary differential equations are usually of order greater than 1. For instance, one of the most usual settings is a mechanical problem in which forces act on a body to determine its motion. The equation that governs this system is, as the reader should know, Newton's second law:

$$\vec{a} = m \cdot \vec{F}$$

which is stated as "acceleration is mass times force." If the position of the body at time t is $(x_1(t), \dots, x_n(t))$, then the acceleration vector is $\vec{a} = (\ddot{x}_1(t), \dots, \ddot{x}_n(t))$ and if the resultant force is $(F_1, \dots, F_n)$ (which would depend on $(x_1, \dots, x_n)$) then Newton's Law becomes,

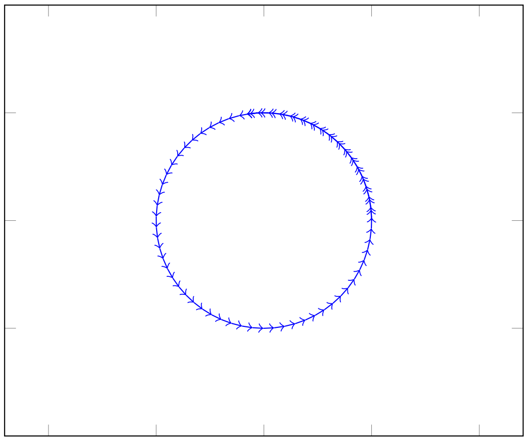

FIGURE 3. Approximation to the solution of Problem (23) using Heun's method. The error is unnoticeable on the printed page.

in coordinates:

$$\begin{cases} \ddot{x}_1(t) = F_1(x_1(t),\dots,x_n(t))\cdot m \\ \ddot{x}_2(t) = F_2(x_1(t),\dots,x_n(t))\cdot m \\ \vdots \\ \ddot{x}_n(t) = F_n(x_1(t),\dots,x_n(t))\cdot m \end{cases}$$

which is an ordinary differential equation but has order greater than one so that we cannot apply the methods explained above straightaway. Notice that *the system is autnomous* because Newton's second law is time-invariant (it depends only on the positions of the bodies, not on time).

However, one can look at a body in motion under the influence of forces as a system whose "state" is not just the position —that is, the set of coordinates $(x_1(t),\dots,x_n(t))$— but as a system whose state is described by position *and* velocity. As a matter of fact, everybody knows by experience that the trajectory of a body (say, a stone) influenced by gravity depends not only on its initial position (from where it is thrown away) but also on its initial velocity (how fast it is thrown away). So, in mechanics, *velocities* are intrinsic elements of the system of coordinates. Thus, the body B under study has $2n$ coordinates to account for its state: its position $(x_1,\dots,x_n)$ and its velocity $(v_1,\dots,v_n)$. By their own nature, these $2n$ coordinates satisfy the following equations:

$$\dot{x}_1(t) = v_1(t),\ \dot{x}_2(t) = v_2(t),\ \dots,\ \dot{x}_n(t) = v_n(t).$$

And now Newton's law can be expressed as a true ordinary differential equation of order one

$$
\begin{cases}
\dot{x}_1(t) = v_1(t) \\
\dot{v}_1(t) = F_1(x_1(t), \dots, x_n(t)) \cdot m \\
\dot{x}_2(t) = v_2(t) \\
\dot{v}_2(t) = F_2(x_1(t), \dots, x_n(t)) \cdot m \\
\vdots \\
\dot{x}_n(t) = v_n(t) \\
\dot{v}_n(t) = F_n(x_1(t), \dots, x_n(t)) \cdot m
\end{cases} \tag{26}
$$

to which the methods explained above can be applied. Notice that one is usually only interested in knowing the positions $(x_1(t), \dots, x_n(t))$ but one *must* solve the system for both positions and velocities. After computing both values, one may discard the latter, if it is truly unnecessary.

For an initial value problem, one needs (as can be seen from Equation 26) not only the position at time t_0 but also the velocity at that time.

3.1. The general case is similar. In general, one starts with an ordinary differential equation of some order k —the maximum order of derivation that appears for some variable and transforms the problem having n variables and the derivatives up to order $k-1$ into one having kn variables by adding $(k-1)$ new variables.

One may state the general case as a differential equation

$$
\begin{cases}
\dfrac{d^k x_1}{dt^k} = F_1(t, \mathbf{x}, \mathbf{x}', \dots, \mathbf{x}^{k-1)}) \\
\dfrac{d^k x_2}{dt^k} = F_2(t, \mathbf{x}, \mathbf{x}', \dots, \mathbf{x}^{k-1)}) \\
\vdots \\
\dfrac{d^k x_n}{dt^k} = F_n(t, \mathbf{x}, \mathbf{x}', \dots, \mathbf{x}^{k-1)})
\end{cases} \tag{27}
$$

where the functions $F_1, \dots, F_n$ depend on t and the variables $x_1, \dots, x_n$ and their derivatives with respect to t up to order $k-1$. It is easier to understand than to write down.

In order to turn it into a system of equations of order one, one just defines $k-1$ new systems of variables, which we enumerate with superindices

$$
(u_1^1, \dots, u_n^1), \dots, (u_1^{k-1}, \dots, u_n^{k-1})
$$

and specify that each system corresponds to the derivative of the previous one (and the first one is the derivative of the coordinates),

$$\left\{\begin{array}{l} \dot{x}_1 = u_1^1 \\ \vdots \\ \dot{x}_n = u_n^1 \end{array}\right., \left\{\begin{array}{l} \dot{u}_1^1 = u_1^2 \\ \vdots \\ \dot{u}_n^1 = u_n^2 \end{array}\right., \dots, \left\{\begin{array}{l} \dot{u}_1^{k-2} = u_1^{k-1} \\ \vdots \\ \dot{u}_n^{k-2} = u_n^{k-1} \end{array}\right.$$

Using this technique, one ends up with a (very long) ordinary differential equation of order one in the variables

$$(x_1, \dots, x_n, u_1^1, \dots, u_n^1, \dots, u_1^{k-1}, \dots, u_n^{k-1}).$$

3.2. The two-body problem. As an example, let us consider a typical problem of newtonian mechanics, the motion of a system of two bodies —which is, again, autonomous. Let B_1 and B_2 be two dimensionless objects with masses m_1 and m_2 which we suppose are placed on the same plane (so that we only need two coordinates to describe their positions). The usual approach is to consider one of the bodies fixed and the other movable but we want to show how both bodies do move and how they do interact with each other. If (x_1, y_1) and (x_2, y_2) denote the coordinates of each body, and dots indicate derivation with respect to the time variable, Newton's equations of motion state that, respectively:

$$\begin{aligned}(\ddot{x}_1, \ddot{y}_1) &= G\frac{-m_2}{((x_2-x_1)^2+(y_2-y_1)^2)^{3/2}}(x_2-x_1, y_2-y_1)\\ (\ddot{x}_2, \ddot{y}_2) &= G\frac{-m_1}{((x_2-x_1)^2+(y_2-y_1)^2)^{3/2}}(x_1-x_2, y_1-y_2)\end{aligned} \tag{28}$$

where G is the gravitational constant. We shall use units in which $G = 1$ in order to simplify the exposition. There are 4 equations in (28), one for each second derivative. Notice that the first derivative of each x_i, y_i does not appear explicitely but this is irrelevant: the problem is of order two and to turn it into one of order one, they have to be made explicit. Given that there are four coordinates, we need another four variables, one for each first derivative. Let us call them (u_x, u_y) and (v_x, v_y). Writing all the equations one after the other for $G = 1$, we get

$$
\begin{aligned}
\dot{x}_1 &= u_x \\
\dot{u}_x &= \frac{-m_2}{((x_2-x_1)^2+(y_2-y_1)^2)^{3/2}}(x_2-x_1) \\
\dot{y}_1 &= u_y \\
\dot{u}_y &= \frac{-m_2}{((x_2-x_1)^2+(y_2-y_1)^2)^{3/2}}(y_2-y_1) \\
\dot{x}_2 &= v_x \\
\dot{v}_x &= \frac{-m_1}{((x_2-x_1)^2+(y_2-y_1)^2)^{3/2}}(x_1-x_2) \\
\dot{y}_2 &= v_y \\
\dot{v}_y &= \frac{-m_1}{((x_2-x_1)^2+(y_2-y_1)^2)^{3/2}}(y_1-y_2)
\end{aligned}
\tag{29}
$$

which is a standard ordinary differential equation of the first order. An initial value problem would require both a pair of initial positions (for (x_1, y_1) and (x_2, y_2)) and a pair of initial velocities (for (u_x, u_y) and (v_x, v_y)): this gives eight values, apart from the masses, certainly.

In Listing 7.1 we show an implementation of Heun's method for the two-body problem (with $G = 1$). In order to use it, the function `twobody` must receive the values of the masses, a vector of initial conditions as $(x_1(0), y_1(0), x_2(0), y_2(0), u_x(0), u_y(0), v_x(0), v_y(0))$, a step size h and the number of steps to compute.

```
% two-body problem with Heun
% x is the initial value vector:
% (x1, y1, x2, y2, v1x, v1y, v2x, v2y)
function s = twobody (m1, m2, x, h, steps)
% use next line for plotting to pngs (prevents plotting on screen)
% set(0, 'defaultfigurevisible', 'on');
% hold on
s = [x(1) x(2) x(3) x(4)];
for k = 1:steps
d = ((x(3) - x(1))^2 + (x(4) - x(2))^2)^(1/2);
F1  = m2/d^3*[x(3)-x(1) x(4)-x(2)];
F2  = -m1/(d^3*m2)*F1;
w1  = [x(5:end) F1 F2];
z   = x + h*[x(5:end) F1 F2];
dd  = ((z(3) - z(1))^2 + (z(4) - z(2))^2)^(1/2);
FF1 = m2/dd^3*[z(3)-z(1) z(4)-z(2)];
FF2 = -m1/(dd^3*m2)*FF1;
w2  = [z(5:end) FF1 FF2];
v   = (w1 + w2)/2;
x   = x + h*v;
s   = [s; x(1:4)];
% next lines for filming, if desired
```

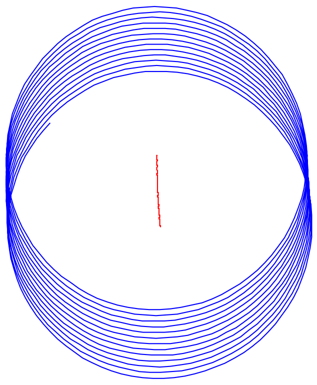

FIGURE 4. Approximation to the solution of the two-body problem with initial values as in the text. One of the bodies (in red) is more massive than the other.

```
%plot(x1,y1,'r', x2,y2,'b');
%axis([-2 2 -2 2]);
%filename=sprintf('pngs/%05d.png',k);
%print(filename);
%clf;
end
endfunction
% interesting values:
% r=twobody(4,400,[-1,0,1,0,0,14,0,-0.1],.01,40000); (pretty singular)
% r=twobody(1,1,[-1,0,1,0,0,0.3,0,-0.5],.01,10000); (strange->loss of
    precision!)
% r=twobody(1,900,[-30,0,1,0,0,2.25,0,0],.01,10000); (like sun-earth)
% r=twobody(1, 333000, [149600,0,0,0,0,1,0,0], 100, 3650); (earth-sun)...
```

LISTING 7.1. An implementation of the two-body problem with $G = 1$ using Heun's method.

In Figure 4, a plot for the run of the function `twobbody` for masses $4, 400$ and initial conditions $(-1, 0)$, $(1, 0)$, $(0, 14)$, $(0, -0.1)$ is given, using $h = 0.01$ and 40000 steps. The difference in mass is what makes the trajectories so different from one another, more than the difference in initial speed. Notice that the blue particle turns elliptically around the red one.

Finally, in Figure 5, we see both the plot of the motion of two particles and, next to it, the relative motion of one of them with respect to the other, in order to give an idea of:

- The fact that trajectories of one body with respect to the other are ellipses.
- The error incurred in the approximation (if the solution were exact, the plot on the right would be a closed ellipse, not the open trajectory shown).

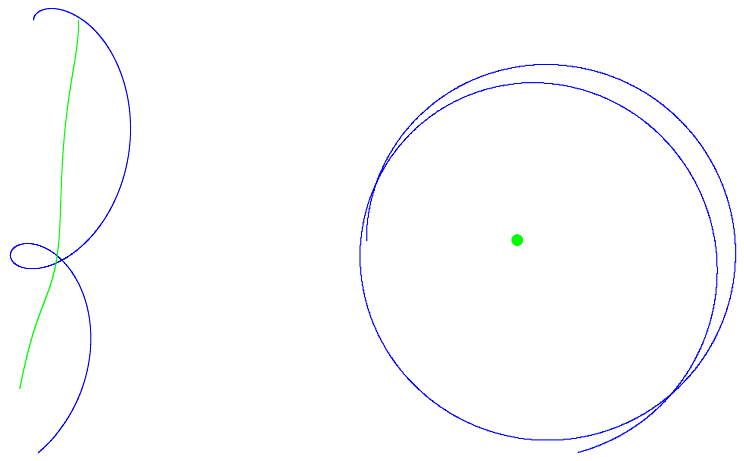

FIGURE 5. Absolute and relative motion of two particles. On the left, the absolute trajectories, on the right, the relative ones.