Methods of Computational Physics

What we would have wanted to know before starting our PhDs and they didn't tell us ...

by Dimitri Laveder and Gilles Niccolini

Version 0.1, November 19, 2012

Copyright ©2011-2012 Dimitri Laveder and Gilles Niccolini. All rights reserved.

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

Preface

When you are teaching, there always comes a time when you know that you have to sit down in front of your computer and start LaTeXising your lecture notes. You know it, but you don't want to do it because it is very time consuming and you've got most certainly more "important" things to deal with ... Well, indeed you know you are wrong because each time you did it in the past it was a great experience: for you and your students. For you, because this exercice forces you to precise your thoughts and clarified them; for your students because the lecture notes you produce provide a reference, and more material than you can present during the class.

Contents

Chapter 1

Introduction

The aim of these lecture notes is to provide the reader with sufficient information to solve the problems of mathematical physics from a numerical point of view. We will basically focus on the resolution of Ordinary Differential Equation (ODEs) and Partial Differential Equations (PDEs). Of course, the subjects covered by computational physics or numerical analysis at large are not restricted to differential equations. This choice is a matter of taste: we do think that solving ODEs and PDEs with a computer is fun.

We will not enter into much details of numerical analysis and our presentation will be guided by the purpose of implementating the methods in the form of a computer program as a final product. Nevertheless, we will try and provide proofs of any results that we might need rather in a physicist style than a mathematician style, say with lack of all the rigour but most of it [1].

We will not treat the subject of computer program development, by lack of place, and consequently these notes will be as far as possible computer language agnostic as far as the presentation of algorithms and mathematics are concerned. When useful, we might use some `C/C++` or `Fortran` short programs within the core of the text but trying to avoid advanced features of theses languages and mostly used them as pseudo-codes. Well, we could have invented some kind of pseudo-code language for the only purpose of being used in these notes but why re-invent the wheel after all ?

Some of our colleagues does not really understand why we bother ourselves coding in `C/C++` or `Fortran` all day long when programs like *Mathlab* or *Mathematica* or alike can do the job. Sometimes your problems are so specific that you need to tune your methods and algorithms to your particular problems and you will not find tools that will do the job without pain. It is almost always impossible to get analytical results for the problem we need to solve in astrophysics such as radiative transfer, radiation hydrodynamics or magneto hydrodynamics, ... There is even no needs to invoke

[1] Yes, let's put it that way.

such difficult physical problems, you might consider these simple differential equations where we want to seek the function y of the variable x

$$y'(x) = x^2 + y^2(x) \qquad \text{or} \tag{1.1}$$

$$y''(x) = 6\,y^2(x) + x \ , \tag{1.2}$$

where it can be proved that no analytical solution can be found (e.g. see Henrici 1962).

Getting analytical results is possible from time to time, but even in that case you might (always ?) want to get *numerical* results out of your *analytical* results. At least this is what you need if these results are to be compared to what Nature has to say about your own particular way of discribing it. The results can be quite involved with series of series of nightmarish transcendental functions. An example is given by the Mie theory of scattering of light by spherical particules (e.g. such as interstellar dust grains or simply water clouds in our atmosphere). You might want to have a look at the Mie theory results in Bohren & Huffman (1998). So ok, we have analytical results and now what ? Well, in this particular case what these authors did is to provide `fortran` codes to numerically compute the results. By the way, these are the kind of analytical results you cannot expect that a formal solver program can find easily, if at all. What if your analytical results are simple, say as simple as a good old $\sin(x)$ for a given real number x ? Well, if you need the numerical value of the sin for a given x, you cannot escape to devise (or let a calculator do it for you) some numerical methods to estimate it to a certain level of precision.

In these lectures, we will mainly expose the mathematics of the numerical methods to solve the equation of mathematical physics, but keeping in mind that the actual computation will be executed by a computer instructed through a computer program. Of course, numerical methods where developed before the advent of the first computers (roughly around 1940). If we look at the name of some famous contributors to numerical analysis such as Gauss (1777), Newton (1643), Lagrange (1736), Runge (1856), etc, and we can even track down to the computation of π (Egyptian, ~ -2000) or $\sqrt{2}$ (Babylonian ~ -1800). Consequently, there is no ambiguity: it has nothing to do with computers after all and the computation could be in principle done by hand. Nowadays, the numerical computation are so complex (in particular in astrophysics) that we need a computer to store the amount of necessary data and to compute as fast as possible. Super computing facilities can reach the PFLOPs, i.e. 10^{12} floating point operations per seconds, but even with these performance, some computation might take days or weeks. In some very specific cases, the TFLOPs can be obtained at almost no cost (well, $\sim$ 2000 euros) compared to large facilities, by using the Graphics Processing Units (GPUs) for general purpose computation. Nowadays, numerical computation, and particularly in astrophysics, are intimately associated with the use of computers and their programming and chances are that if you start a phd you finally end up using a computer to do some computations.

Chapter 2

Representation of numbers

2.1 Introduction

In this chapter we will give a brief description on how real number are represented in order to be handled by computers. We will not enter into the most gory details of the arithmetic of floating point numbers. The reader can find them in numerous reference but for instance Goldberg (1991) is a very good start.

The good news is that most of the time we can just ignore that we are not dealing with real numbers but just a finite subset of it: the floating point numbers. The level of precision to which we can represent real number and proceed to some basic arithmetic operation is of course finite but, in the so called `double` precision IEEE standard used my almost every computer on earth, is so tiny, 10^{-16}, that we can most of the time ignore its effect. Most of the time, but not every time. You will eventually put yourself into uncomfortable situations where for instance

$$(a+b)+c \neq a+(b+c) \ , \tag{2.1}$$

$$\exp(\log x) \neq x \ , \tag{2.2}$$

$$x^2 - y^2 \neq (x-y)\,(x+y) \ , \tag{2.3}$$

$$1 + \underbrace{10^{-17} + 10^{-17} \cdots 10^{-17}}_{10^{17}} \neq 2 \ , \tag{2.4}$$

and you might conclude that we live in a dangerous world. However, the situation might look even worse considering the following example. If we think it twice, we all know that 0.1 cannot be represented exactly in base 2. Indeed, the representation of 0.1 is $0.0001100110011\ldots$. So, at some point we have to stop our notation and we make consequently an error because of this truncation. It is pretty much the same as considering that $\frac{1}{3}$ as an infinite decimal representation $0.33333\ldots$, except that we are more used to it. In your favourite computer language, whenever you write something like `double x=0.1;` for instance, you make an error. And if you know how to print the 50 first digits (in base 10) representing this number (e.g. try `printf("%.50e\n",0.1);` in `C`), you will find that indeed x is rather given by

$$x \approx 0.1000000000000000055511151231257827021181583404541 02\ldots\ ,$$

clearly not what you expected.

Consider the following `C++` program (not a very difficult one, but just skip this if you cannot understand it)

```cpp
#include <iostream>
#include <cstdlib>

using namespace std;

int main() {
  double a=1.;
  double b=10.;
  double c=a/b;

  if(c==a/b)
    cout << "Ok" << endl;
  else
    cout << "Not ok ... :-)" << endl;

  return EXIT_SUCCESS;
}
```

You might think that the results of this program would be to always print out “Ok”. You are wrong. It’s a compiler issue (`gcc` in that case) but definitely not a bug, say, an obscure feature. At some point during the compilation, `c` and `a/b` are not computed with the same level of accuracy (a compiler feature) and since 0.1 cannot be represented exactly, you do not know what is the results, it depends on the version of the compiler and your system. It might be the case that the compiler does not create a variable, i.e. a location in the data of your program, for storing the results of `a/b` but instead uses a processor register.

This is a very strange behaviour and you cannot understand it if you ignore the way numbers are represented by computers.

2.2 Floating point numbers

Let’s start with the usual decimal (Arabic) notation of numbers like 1.34, 3.14 or 0.9… where the dots represent the infinite repetition of 9. For $x \in \mathbb{R}$ we can always write x in the following manner

$$x = \left(\sum_{k=0}^{\infty} d_{k+1}\, 10^{-k} \right) \times 10^{e}\ , \qquad (2.5)$$

where we can take $d_{k+1} = \lfloor x \times 10^{k-e} \rfloor - \lfloor x \times 10^{k-e-1} \rfloor\ 10$ and $e = \lfloor \log_{10}(x) \rfloor$ for $k \geq 0$.

However, when you write a number down in this form, you usually truncate the notation and you write something like this

$$x = d_1.d_2 d_3 \cdots d_p \times 10^e \ , \tag{2.6}$$

using the "scientific notation" with an exponent e and p digits in base 10.

We can formalise this to obtain the *floating point numbers*, which are of this form. x is a floating point number if

$$x = \pm m \times \beta^e \ , \tag{2.7}$$

where m, the *mantissa*, can be written in the following form : $m = 0.d_1 d_2 \cdots d_p$ with *only* p digits.

With such a definition there is no unique notation; we could write 0.3×10^1 or 0.03×10^2 for instance. In order to ensure the uniqueness we impose that $d_1 \neq 0$ and the number are said to be normalized. Of course in that way 0 is simply missing and we add it to the set of floating point numbers. A system of floating point numbers is defined by the number of digits to be used, the base $\beta > 2$, and the max and min value for the exponent, $e_{\max}$ and $e_{\min}$ respectively. Consequently, there are $2 \times (\beta - 1) \times \beta^{p-1} \times (e_{\max} - e_{\min} + 1) + 1$ floating point numbers in a given system.

A mantissa m verifies the following inequalities

$$0.1\underbrace{0 \cdots 0}_{p-1} \leq m \leq 0.\underbrace{(\beta-1) \cdots (\beta-1)}_{p} \ , \tag{2.8}$$

that simplifies in

$$\beta^{-1} \leq m \leq 1 - \beta^{-p} \ . \tag{2.9}$$

In the following, for the seek of simplicity, we will use a strict inequalities on the right, i.e.

$$\beta^{-1} \leq m < 1 \ . \tag{2.10}$$

2.3 Rounding operation

2.3.1 Definition

Let's consider a *real* mantissa m that quite generally will not be a floating point number (from now on, we will use the shortest term "floats"). If we want to represent this mantissa as a float, we have to discard some of the digits and keep only p of them. We could indeed simply remove them. This operation is called "truncation". However, most of the time what we want to do is to choose the float that is the closest to m. This operation is called rounding to nearest, or simply rounding. By writing down a few examples, you can convince yourself that this operation can be formalised as

$$m_p \hat{=} fl_p(m) = \beta^{-p} \left\lfloor \beta^p\, m + \frac{1}{2} \right\rfloor \qquad \text{for } m > 0 \text{ and} \tag{2.11}$$

$$= \beta^{-p} \left\lceil \beta^p\, m - \frac{1}{2} \right\rceil \qquad \text{for } m < 0 \;, \tag{2.12}$$

where $\lfloor x \rfloor$ and $\lceil x \rceil$ are respectively the floor and the ceiling (i.e. integer parts) functions.

2.3.2 Errors

By approximating m by m_p we make an error, the so-called *rounding error*. Let's derive an upper limit to the rounding error. From the definition of $\lfloor x \rfloor = \max\{i \in \mathbb{Z} \mid i \leq x\}$, we have

$$x - 1 < \lfloor x \rfloor \leq x \;. \tag{2.13}$$

From Eq. (2.11) and (2.13) we derive

$$\beta^p\, m - \frac{1}{2} < \left\lfloor \beta^p\, m + \frac{1}{2} \right\rfloor \leq \beta^p\, m + \frac{1}{2} \;,$$

and multiplying this last relation by β^{-p} we obtain

$$m - \frac{1}{2}\,\beta^{-p} < m_p \leq m + \frac{1}{2}\,\beta^{-p}$$

and by definition of the *absolute* rounding error $\Delta m = m_p - m$ we finally obtain

$$|\Delta m| \leq \frac{1}{2}\,\beta^{-p} \;. \tag{2.14}$$

This upper limit is actually $\frac{1}{2}$ *units in the last place* (the p^{th} digits after the ".") or *ulps*.

Recalling Eq. (2.10), since $\frac{1}{m} \leq \beta$, we finally have for the *relative* rounding error

$$\left| \frac{\Delta m}{m} \right| \leq \frac{\beta}{2}\,\beta^{-p} \;. \tag{2.15}$$

The upper limit in (2.15) is called the *rounding unit* and is noted $u = \frac{\beta}{2}\,\beta^{-p}$.

2.4 Floating point operation

When we want to represent a real number by a float, we have seen that we can introduce an error which can be as large as one rounding unit. Is this the only source of error ? The answer is negative. When we proceed to a basic arithmetic operation between floats, it might happen that the results is not a float and may require more digits to be stored than we are allowed to or simply an infinite number of digit (consider the division of 1 by 10 in base 2). We will use the notation $\oplus$, $\ominus$, $\otimes$ and $\oslash$ for the floating point arithmetic operation. For instance, we cannot assume $+$ and $\oplus$ to be equal and in general $x + y \neq x \oplus y$ *even if* x and y are indeed floats.

We can adopt a very simple model of the floating point operation. We will forget about the details of the hardware implementation of the actual arithmetic operation and assume that we can have as many bits as necessary to store the result of the operations on two mantissas and then still get the result correctly rounded. We summarise the procedure as follows

1. Before doing the actual operation, mantissa might be aligned (for addition and subtraction) , the exponent changed accordingly or added/-subtracted for multiplication and division,
2. the operation is carried out and the results stored in a sufficiently large register,
3. we might shift again the mantissa of the results in order to have a normalized float,
4. then we results is rounded with the correct number of digits

Indeed, following this procedure we only introduce an error in the last item of the above list when we correctly round the results. However, we already know the relative error we might introduce, 1 rounding unit. Sticking to this very simplistic model there nothing more to add and we have the following results for operation *op* (standing for $+$, $-$, $\times$ and $/$)

$$x \bigcirc\!\!\!\!\!\!\text{op}\; y = x \,\text{op}\, y\,(1+\delta) \ , \tag{2.16}$$

with $|\delta| \leq u$.

Note that the above results is not necessarily ensured if we do not have access to a sufficiently large register to store the results. Let's figure out the number of bits the register must contain.

For the multiplication of two floating point mantissas $m_x = 0.d_1d_2d_3\cdots d_p$ and $m_y = 0.d'_1d'_2d'_3\cdots d'_p$ obviously the results can be *exactly* stored with $2p$ digits. However, we do not need to store $2p$ digits because we will round the results to p digits anyway. If we consider the product of the 2 mantissas we have

$$\beta^{-2} \leq m_x\, m_y \leq 1 \ ,$$

and we see that we might need to shift the results at most 1 digit to the left to normalized it. Basically, when we need this normalisation (i.e. when

$m_x\, m_y \leq \beta^{-1}$) the results is of the form

$$m_x\, m_y = 0.0\,\underbrace{d''_1 d''_2 \cdots d''_{p+1}}_{p+1 \text{ digits}} \cdots \ .$$

Now, when we shift to the left to normalized, the p first digits of the results $d''_1 \cdots d''_p$ will get in the right place *but* we need an extra digit to round correctly the number to p digits. The conclusion is that finally we need only to store the results with $p+2$ digits in order to get the correctly rounded results. For the division the situation is pretty much the same if we consider not $\frac{m_x}{m_y}$ but $\frac{\beta^{-1}\, m_x}{m_Y}$ instead while updating the exponent accordingly. We just have to notice that in some situation the result just cannot be stored exactly (and $2p$ digits will not be sufficient to store it) but we again only need $p+2$ digits to get the correctly rounded results for the same reasons.

For the addition and subtraction the picture darkens somehow. If we consider two normalized mantissas, expressed in base 2 for the sake of simplicity, the least difference that we can get is given by

$$\begin{aligned} |m_x - m_y| &= 0.\underbrace{10\cdots 0}_{p}0 \\ &- 0.0\underbrace{11\cdots 1}_{p} \ . \end{aligned}$$

Generalising to an arbitrary base β, we see that the lower bound on $|m_x - m_y|$ is $\beta^{-1} - \beta^{-1}\ (\beta - 1) \sum_{k=1}^{p} \beta^{-k}$ or simply $\beta^{-(p+1)}$. This means that we might end up in situations where the difference $|m_x - m_y|$ is given by

$$|m_x - m_y| = 0.\underbrace{00\cdots 0}_{p} d''_1 d''_2 \cdots d''_p d''_{p+1} \ , \tag{2.17}$$

from which we see that to correctly round the results after normalisation, a shift of p digits to the left, we need $2p+1$ digits.

The conclusion is that within the limit of our simple floating point operation model, we need globally to have a register as large as $2p+1$ digits. Typically, the register are nowadays 80 bits large with actual processors. Note that with IEEE double precision (see next section) we unfortunately need 105 digits. In fact, it can be shown that in base 2 only $p+2$ bits are necessary to get the correctly rounded results(see Knuth 1981). Of course, the actual implementation of the arithmetical operations get a bit more complicated (the more bits to store the intermediary results the less difficult to implement) but the main point is that it can be achieved and as far as arithmetical operations are concerned you can rely on Eq. (2.16) to be true.

2.5 IEEE floating point standard

The Institute of Electrical Electronics Enginners (IEEE) defined a standard for the floating point numbers. There are several rounding modes we did not talk about, but in particular if we round to the nearest floating point number with (2.11) or (2.12) the standard ensures that the upper limit to the relative error is indeed 1 rounding unit. Concerning the arithmetic operation we also have

$$x \bigcirc\!\!\!\!\!\!\;\mathit{op}\; y = x\, op\, y \times (1+\delta) \ , \tag{2.18}$$

where δ, the relative error, must verify

$$|\delta| \leq \frac{\beta}{2}\beta^{-p} = u \ . \tag{2.19}$$

There is nothing imposed on usual functions such as sin, exp, log, etc, except for the squared root for which we must have

$$\mathrm{sqrt}(x) = \sqrt{x} \times (1+\delta) \ , \tag{2.20}$$

with again $|\delta| \leq \frac{\beta}{2}\beta^{-p}$.

2.5.1 Single precision

In the single precision (“`float`” in C/C++) specifications we use 32 bits words to code the number, we have

- 1 for the bit sign,
- 23 for the mantissa and
- 8 for the exponent.

In this system, the mantissa are given by $d_1.d_2d_3\cdots d_p$, slightly different from our convention. An interesting point is that $d_1 \neq 0$, in base 2 it is always 1. In practice we will not use a bit for setting it to 1, we would simply lose that bit. Actually, we can assume that $d_1 = 1$ and use p bits after the “decimal” point. We gain an extra bit of precision with this procedure and we have $p = 24$, not 23 !

From the previous considerations, the rounding unit (the machine precision actually) is equal to $2^{-24} \simeq 5.96046 \times 10^{-8}$.

We can defined the so-called *machine epsilon*, ϵ, sometimes abusively called the machine precision[1], as the distance between 1 and the first floating point number immediately larger than 1. In this case it is 2^{p-1} with $p = 24$

[1]Recall that the rounding unit is the machine precision.

i.e. $\epsilon = 1.19209 \times 10^{-7}$. Indeed $\epsilon = 2\,u$, but if we consider the precision to which numbers are rounded independently of the rounding mode (rounding to nearest as we did or rounding by excess or by default), in the worst case it will indeed be the machine epsilon. In practice, we will always use the rounding to nearest mode so that the machine precision is indeed half ϵ.

We have 2^8 possibilities to encode the exponent. However, the sequence of bits corresponding to 00000000 and 11111111 are reserved. The first, to code 0 and denormalized numbers (yes, they exist finally but we will ignore them . . .) and the former to code exceptions ($\pm\infty$, NaN, i.e. Not A Number, etc) when we feel like dividing by 0 for instance. So we still have $2^8 - 2 = 254$ possible exponents. It would not be wise to use a bit just to code the sign of the exponent. Instead, the standard code the value of the exponent plus $2^7 - 1 = 127$, i.e.

$$e + 127 = 1, \cdots, 254 \ , \tag{2.21}$$

or

$$e = -126, \cdots, +127 \ . \tag{2.22}$$

The largest normalized float is then $\max = 2 \times 2^{127} \simeq 3.40283 \times 10^{38}$ (when all the mantissa bits are set to 1) and the smallest $\min = 2^{-126} \simeq 1.17549 \times 10^{-38}$. If we obtain a number lower than min that cannot be represented as normalized float, the situation is called *floating underflow*. Conversely, if a number is larger than max we are dealing with a *floating overflow*.

To summarise we have

$$\begin{aligned} u &\simeq 5.96046 \times 10^{-8} \ , \\ \epsilon &\simeq 1.19209 \times 10^{-7} \ , \\ \min &\simeq 1.17549 \times 10^{-38} \qquad \text{and} \\ \max &\simeq 3.40283 \times 10^{38} \ . \end{aligned} \tag{2.23}$$

2.5.2 Double precision

For the double precision (“`double`” in C/C++) a 64 bits word is used with 1 bit for the sign, 52 for the mantissa (getting an extra bit of precision with the same trick, i.e. $p = 53$), and 11 for the exponent. We leave it as an exercise for the reader to show that we have

$$\begin{aligned} u &\simeq 1.11022 \times 10^{-16} \ , \\ \epsilon &\simeq 2.22045 \times 10^{-16} \ , \\ \min &\simeq 2.22507 \times 10^{-308} \qquad \text{and} \\ \max &\simeq 1.79769 \times 10^{308} \ . \end{aligned} \tag{2.24}$$

Just a last remark. It is not strictly speaking true to say that in C/C++ `float` and `double` correspond to IEEE single and double precision. It will be true in practice in most of cases, but the C/C++ standard just says that the size (in bits) of a `double` is larger or equal to the size of a `float`. However, in Fortran it is possible to specify exactly what you want with `real(kind=4)` (4 bytes) or `real(kind=8)` (8 bytes).

Chapter 3

Mathematical tools

3.1 Interpolation

To fix the ideas, let's assume that we are given a set of real abscissa x_i for $i = 1, \cdots, n$. Let's further assume that we have a functional relation between the x values and another real variable y by the mean of a real function f, and that the only information we have is the values of $y_i = f(x_i)$. What if we need to make a guess on the value of $y = f(x)$ for a given x different from the x_i ? If x belongs to the interval spanned by the x_i's, this is called *interpolation*, outsite it is called *extrapolation*.

When do we want to do such a thing ? Perhaps because we have some experimental or numerical data and we need to make a guess between the tabulated values of a given function. Before the rise of computer, in order to determine the value of a special function, like this good old "sin" for instance, you would interpolate from the tabulated values in a handbook (see Abramowitz & Stegun 1964) for instance. Nowadays or course, you might want to use a computer program to get these results.

Undoubtedly, interpolated tabulated values is often quite useful. Our interest here is of a different kind though. Interpolation is the basis of numerous methods in numerical analysis. In particular, we will make use of interpolation when trying to solve differential equation or simply integrating real functions. The reason is obvious: we live in a discrete world. We will solve the problems of continuous mathematics by making them discrete and there is no way to escape that. For instance, if we want to play with a real function f of a real variable x we will explicitly, but most of the time implicitly, use tabulated values of this functions in our considerations in order to differentiate or integrate this function.

In summary, our interest in the interpolation is not practical-we don't want to interpolate data- but theoretical since it is inherent to almost all the numerical methods presented in theses lecture notes.

3.1.1 Lagrange interpolation

Again, we are provided with two sets of values: one for the abscissas, the x_i's and the corresponding ordinate values, the $y_i = f(x_i)$ where f is a given function. Notice that we do not have necessarily access to f (for instance when dealing with experimental data). The aim of the game is to estimate the value of $f(x)$ for $x \neq x_i$ and provide an expression to the error we make in this procedure.

Our estimation must be a functional of the y_i and a function of x. One of the simplest thing we can do is to use a linear combination of the $f(x_i)$'s. If $\bar{f}(x)$ is the approximation to $f(x)$ we are seeking for we can use

$$f(x) = \underbrace{\sum_{k=1}^{n} l_k(x)\, f(x_k)}_{\bar{f}(x)} + e(x) \tag{3.1}$$

where the $l_k(x)$ are real coefficients depending on x and $e(x)$ is the error.

Up to that point, our considerations are quite general and we did not made any assumption about the l_k function (of x). By the definition of the interpolation procedure we must have $f(x_i) = \bar{f}(x_i)$ for all $i = 1, \cdots n$ or in other words $e(x_i) = 0$. So if we set $x = x_i$ in Eq. (3.1) we obtain

$$f(x_i) = \sum_{k=1}^{n} l_k(x_i)\, f(x_k) \ , \tag{3.2}$$

from which we conclude that we must have

$$l_k(x_i) = \delta_{ki} \ , \tag{3.3}$$

where δ is the *Kronecker* symbol.

Eq. (3.2) is a general linear [1] interpolation formula. We need to specify the expression of the $l_k(x)$ functions in order to obtain something useful.

With *Lagrange* interpolation we chose polynomials for the l_k's. Indeed, through the n points (x_i, y_i) for $i = 1, \cdots, n$ goes a unique polynomial $\bar{f}$ of degree $n-1$. $\bar{f}$ being a linear combination of the l_k's, there are themselves polynomials of order $n-1$. From Eq. (3.3) we know that l_k has the $n-1$ roots x_i for $i \neq k$ and that $l_k(x_k) = 1$. From these properties l_k can be expressed as

$$l_k(x) = \frac{(x-x_1)\,(x-x_2)\,\cdots\,(x-x_{k-1})\,(x-x_{k+1})\cdots\,(x-x_n)}{(x_x-x_1)\,(x_k-x_2)\,\cdots\,(x_k-x_{k-1})\,(x_k-x_{k+1})\cdots\,(x_k-x_n)} \ , \tag{3.4}$$

[1] Linear in the sense that we mix up the $f(x_i)$ linearly, but this does not mean $\bar{f}$ will be linear.

where we assume that we are not dealing with the particular case $x \neq x_1$ and $x \neq x_n$ for the purpose of showing clearly the avoidance of the $(x - x_k)$ term.

Formally, it is customary to introduce the polynomial $P_n(x)$ defined by

$$P_n(x) = \prod_{i=1}^{n} (x - x_i) \ , \tag{3.5}$$

and plug it in the expression Eq. (3.4) of the l_k's to obtain after a little algebra the following expression

$$l_k(x) = \frac{P_n(x)}{(x - x_k)\ P'_n(x_k)} \ . \tag{3.6}$$

To prove Eq. (3.6) we must remark that the derivative of P_n is simply given by

$$P'_n(x) = \sum_{i=1}^{n} \prod_{\substack{k=1 \\ k \neq i}}^{n} (x - x_k) \ , \tag{3.7}$$

which give the denominator of Eq. (3.4) when we evaluate it at $x = x_k$.

3.1.2 Error expression

We are not through yet. We need an expression of the error $e(x)$ that will allow us to derive upper bounds of its absolute value $|e(x)|$. In order to do so we need a preliminary results.

First, recall the *Rolles* theorem stating that if f is a real function of a real variable x continuous in $[a, b]$ and differentiable on (a, b) verifying $f(a) = f(b) = 0$, then there exists $c \in (a, b)$ such that $f'(c) = 0$.

From the *Rolles* theorem we can deduce by inference that if g, a real function of real variable x, has $n + 1$ zeros $g(z_0) = g(z_1) = \cdots = g(z_n)$ then there exists $\alpha\ in\mathbb{R}$ in the open interval spans by the z_is such that we have

$$g^{(n)}(\alpha) = 0 \ . \tag{3.8}$$

To show this we assume first that the zeros are ordered like this $z_0 < z_1 < \cdots < z_n$. We can apply the *Rolles* theorem on each sub-interval (z_i, z_{i+1}): there exists an α_i in each interval such that $g^{(1)}(\alpha_i) = 0$. The derivative $g^{(1)}$ of g now has n zeros, the α_is. Consequently, we can repeat the process and show by inference the results expressed by Eq. (3.8).

Let's define now the function F of a real variable z (not x!) by

$$F(z) = f(z) - \bar{f}(z) - \left[f(x) - \bar{f}(x)\right] \frac{P_n(z)}{P_n(x)} \ , \tag{3.9}$$

where $\bar{f}(x)$ is the value of the *Lagrange* polynomial at x and where we further assume that $x \neq x_i$ (for $i = 1, \cdots, n$).

We note that F has $n+1$ zeros since $F(x_i) = 0$ but also $F(x) = 0$. Then, we can apply our lemma Eq. (3.8) and there exists an α in the interval span by the x_is and x such that $F^{(n)}(\alpha) = 0$.. First, by remarking that $\bar{f}$ being a polynomial of the n^{th} degree its n^{th} derivative is zero, i.e. $\bar{f}^{(n)}(z) = 0$ for all z. Second, from the definition of $P_n(x)$ Eq. (3.5) we further have $P_n^{(n)}(x) = n!$. Finally, we have obtained the expression of $e(x)$ that we were looking for since $f(x) - \bar{f}(x) = e(x)$

$$e(x) = \frac{P_n(x)}{n!}\, f^{(n)}\left(\alpha(x)\right) \ , \tag{3.10}$$

where we write explicitly the dependence of α on x. We derive Eq. (3.10) assuming that $x \neq x_i$ for any i. However, since for $x = x_i$ we have $P_n(x) = 0$ then Eq. (3.10) is also valid for $x = x_i$.

To summarise our results, the *Lagrange* interpolation is described by thw following formula

$$f(x) = \underbrace{\sum_{k=1}^{n} \frac{P_n(x)}{(x - x_k)\, P_n'(x_k)}}_{\bar{f}(x)} + \underbrace{\frac{P_n(x)}{n!}\, f^{(n)}\left(\alpha(x)\right)}_{e(x)} \ , \tag{3.11}$$

for $\alpha(x)$ in the interval spanned by the x_is and x.

The expression for the error Eq. (3.10) contained an *unknown* function of x: of course, otherwise we would know the error or in other words the value of $f(x)$ itself! Still, this expression is quite useful to derive an upper bound on the error. If we know for instance that $f^{(n)}$ is bounded by a constant M in the interval spanned by the x_i and x we deduce an upper bound on $|e(x)| \leq M\, |\frac{P_n(x)}{n!}|$

3.1.3 Things can go wrong

A few remark on problems we might want to avoid. Even if you know an upper bound M on $|f^{(n)}(x)|$ for all x be very careful: *extrapolation*, whenever x is *outside* the interval spanned by the x_is, is a dangerous business. You can convince yourself by noticing that $\lim\limits_{|x| \to \infty} |P_n(x)| = \infty$. It is the upper bound on $|e(x)|$ that goes to ∞ to $e(x)$ itself so we can only say that things might go wrong. Perhaps they will not, but you might have found out for yourself that the *Murphy*'s law is unfortunately verified very often. Ok, we are not going to interpolate just for the fun of it but for the purpose of approximating continuous differential operator. So extrapolation is not our main concern but we better keep in mind when things might go to hell and avoid a lot of annoyances.

However, there is a problem more critical for us. When n is large, unfortunately not quite large since $n \geq 4$ can be sufficient, the *Lagrange* polynomial has the unfortunate tendency to oscillate. To illustrate this behaviour

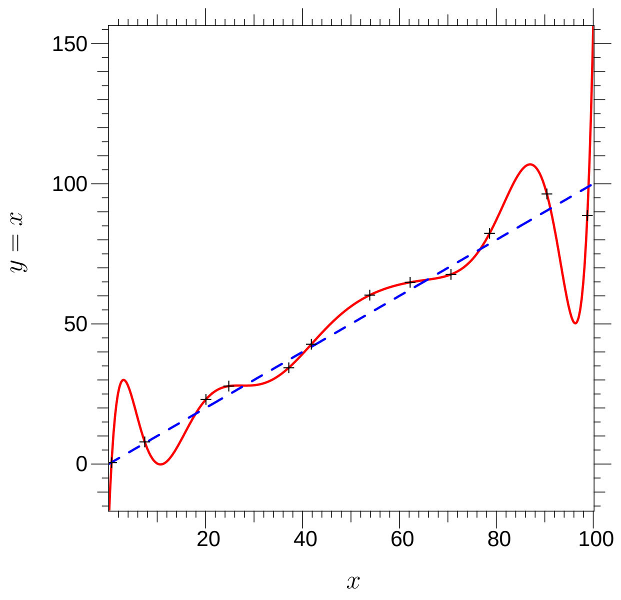

Figure 3.1: Runge effect. The dash line represent the line from which the points (crosses) are generated between $a = 0.1$ and $b = 100$ for $n = 13$ points. In the abscissas and ordinates we introduced some random noise: the x values can be shifted up to half the interval $\frac{b-a}{n-1}$ and a 15 % noise is introduced for the ys values. From a val From this set of points we compute the *Lagrange* polynomial showing the oscillating behaviour.

let's consider the straight line $y = x$. We generate points from this line but we add some noise in the x and y value. For 12 points, we clearly can see the oscillations in Fig. (3.1).

We must be aware of those possible oscillations. When we proceed as in Fig. (3.1), i.e. when we do interpolate some data, by having a look at the *Lagrange* polynomial we can clearly see that we are in trouble. However, when we will use the polynomial to elaborate a numerical method for the integration of differential equation we will not see directly the polynomial we will be using and it could be difficult to identify the problem. Usually, to remedy to this problem we interpolate by block on sub-intervals with polynomial of low degree (3 or 4). For the purpose of interpolating data this is called *spline* interpolation if we further add additional requirement of continuity for the function and its successive derivatives (see Ralston & Rabinowitz 2001, for more details).

3.2 Integration

3.3 Linear system of equations

For solving PDEs we will use finite differences to represent differential operators. In some occasions, in order to obtain our numerical approximation we will need to solve for linear systems of equations of the type

$$\boldsymbol{A}\,\boldsymbol{x} = \boldsymbol{b} \ , \tag{3.12}$$

where $\boldsymbol{x} \in \mathbb{R}^m$ is the unknown, $\boldsymbol{A}$ is a $m \times m$ matrix and $\boldsymbol{b} \in \mathbb{R}^m$ is a given vector.

3.3.1 Matrix splitting methods

The idea behind this kind of methods is to derive an iterative process in which a sequence of estimations, $\boldsymbol{x}^n$, converges to the solution $\boldsymbol{x}$ of Eq. (3.12) as $n \to \infty$. Let's assume that we can *split* the matrix $\boldsymbol{A}$ in two parts $\boldsymbol{A} = \tilde{\boldsymbol{A}} - \Delta\boldsymbol{A}$ where $\tilde{\boldsymbol{A}}$ can be easily inverted. Let's further assume that we already have an estimation $\boldsymbol{x}^n$ and that we want to derive the next (better) estimation $\boldsymbol{x}^{n+1} = \boldsymbol{x}^n + \Delta\boldsymbol{x}^n$, which defines $\Delta\boldsymbol{x}^n$. Ideally, we would like to solve exactly for $\Delta\boldsymbol{x}^n$

$$\left(\tilde{\boldsymbol{A}} - \Delta\boldsymbol{A}\right)\,(\boldsymbol{x}^n + \Delta\boldsymbol{x}^n) = \boldsymbol{b} \ . \tag{3.13}$$

Of course, we assume that we cannot (or don't want) to solve for Eq. (3.13) because it would mean we do not have to use an iterative method at all !

Let's further assume that in some sense $\Delta\boldsymbol{A}$ is "small" compare to $\tilde{\boldsymbol{A}}$ and that "order 2" terms such as the product $\Delta\boldsymbol{A}\,\Delta\boldsymbol{x}^n$ can be neglected compared to other. We expect (hope) that the closer $\tilde{\boldsymbol{A}}$ and $\boldsymbol{A}$ will be, the more efficient our method will be. It seems reasonable to conjecture such a behaviour because in the degenerate case where $\Delta\boldsymbol{A} = \boldsymbol{0}$ our "iterative" method gives the answer in just one single step, the best convergence rate you can dream of ...

Valid or not[2], our approximation (i.e. neglecting $\Delta\boldsymbol{A}\,\Delta\boldsymbol{x}^n$) to Eq. (3.13) suggests the following iterative process

$$\Delta\boldsymbol{x}^n = \tilde{\boldsymbol{A}}^{-1}\,(\boldsymbol{b} - \boldsymbol{A}\,\boldsymbol{x}^n) \ , \tag{3.14}$$

or equivalently

$$\boldsymbol{x}^{n+1} = \tilde{\boldsymbol{A}}^{-1}\,(\boldsymbol{b} + \Delta\boldsymbol{A}\,\boldsymbol{x}^n) \ , \tag{3.15}$$

where we made use of the definition of $\Delta\boldsymbol{x}^n$, i.e. $\boldsymbol{x}^{n+1} = \boldsymbol{x}^n + \Delta x^n$.

[2]Our only concern here is that the resulting iterative method is convergent and efficient !

Obviously, the true solution to the linear system, $\boldsymbol{x}$, verifies Eq. (3.15) and (3.14) which expresses the fact that $\boldsymbol{x}$ is a fixed point of the iterative process. We still cannot say that the methods we are constructing are convergent but at least it seems to make sense to continue our little adventure.

We will have an approach that will certainly make any mathematician bumps his head on the walls because we won't bother too much with the mathematics of the convergence in particular cases. Our approach will be to proceed finger crossed when faced with a given problem and bother about convergence if we really have to do it. For more details you might want to have a look at LeVeque (2007).

At least, here comes a description on how to prove convergence. We introduce the error at iteration step n by $\boldsymbol{e}^n = \boldsymbol{x}^n - \boldsymbol{x}$. By injecting this definition for the iteration step $n+1$ in Eq. (3.15) we obtain

$$\boldsymbol{e}^{n+1} = \tilde{\boldsymbol{A}}^{-1}\,\Delta\boldsymbol{A}\,\boldsymbol{e}^n \ , \tag{3.16}$$

and by inference we can easily show that we have

$$\boldsymbol{e}^n = \left(\tilde{\boldsymbol{A}}^{-1}\,\Delta\boldsymbol{A}\right)^n \boldsymbol{e}^0 \ . \tag{3.17}$$

Now, let's assume that the matrix $\tilde{\boldsymbol{A}}^{-1}\,\Delta\boldsymbol{A}$ can be diagonalize in the following form $\boldsymbol{R}\,\boldsymbol{\Lambda}\,\boldsymbol{R}^{-1}$ where $\boldsymbol{\Lambda}$ is diagonal. The convergence, i.e. $\lim_{n\to\infty}\|\boldsymbol{e}^n\| = 0$ for a given norm is then expressed by

$$\rho\left(\tilde{\boldsymbol{A}}^{-1}\Delta\boldsymbol{A}\right) < 1 \ , \tag{3.18}$$

where $\rho\left(\tilde{\boldsymbol{A}}^{-1}\Delta\boldsymbol{A}\right)$ is the spectral radius of $\tilde{\boldsymbol{A}}^{-1}\Delta\boldsymbol{A}$ defined by $\max_{i=1,\cdots,m}|\lambda_i|$, where the λ_i's are the diagonal elements of $\boldsymbol{\Lambda}$.

Jacobi

The *Jacobi* method is obtained by using the simplest matrix we can invert by taking the diagonal of $\boldsymbol{A}$

$$\tilde{\boldsymbol{A}} = \begin{pmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & \cdots & a_{mm} \end{pmatrix} \ . \tag{3.19}$$

Consequently, the iteration algorithm is given by

$$x_i^{n+1} = \frac{1}{a_{ii}}\left(b_i - \sum_{\substack{j=1\\ j\neq i}}^{m} a_{ij}\,x_j^n\right) \ , \tag{3.20}$$

where x_i^n is the i^{th} element of $\boldsymbol{x}^n$.

Gauss-Seidel

The *Gauss-Seidel* iterative method is obtained by considering the lower part of $\boldsymbol{A}$ including the diagonal, i.e.

$$\tilde{\boldsymbol{A}} = \begin{pmatrix} a_{11} & 0 & \cdots & 0 \\ a_{21} & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mm} \end{pmatrix} , \tag{3.21}$$

since we know how to invert such matrix by forward substitution. Let's assume that we want to solve for $\boldsymbol{y}$ in the linear system $\tilde{\boldsymbol{A}}\,\boldsymbol{y} = \boldsymbol{c}$. We can solve the first equation to obtain

$$y_1 = \frac{b_1}{a_{11}} . \tag{3.22}$$

Then we repeat the process like this

$$\begin{aligned} y_2 &= \frac{1}{a_{22}} \left(c_2 - a_{21}\,y_1\right) , \\ y_3 &= \frac{1}{a_{33}} \left(c_3 - a_{31}\,y_1 - a_{32}\,y_2\right) , \\ y_i &= \frac{1}{a_{ii}} \left(c_i - a_{i1}\,y_1 + a_{i2}\,y_2 + \cdots + a_{i\,i-1}\,y_{i-1}\right) , \\ \vdots &= \vdots \\ y_m &= \frac{1}{a_{mm}} \left(c_m - a_{m1}\,y_1 + a_{m2}\,y_2 + \cdots + a_{m\,m-1}\,y_{m-1}\right) , \end{aligned} \tag{3.23}$$

to obtain the remaining values of y_i for $i = 2, \cdots, m$.

With this algorithm we obtain the solution $\boldsymbol{y}$ to our problem. Note that in Eq. (3.23) we determine the value of y_i from the *already* computed and updated values of y_j for $j < i$.

By remarking that Eq. (3.15) is of the same form with $\boldsymbol{c} = \boldsymbol{b} + \Delta\boldsymbol{A}\,\boldsymbol{x}^n$ and $\boldsymbol{y} = \boldsymbol{x}^{n+1}$, we obtain

$$x_i^{n+1} = \frac{1}{a_{ii}} \left[b_i + \left(\Delta\boldsymbol{A}\,\boldsymbol{x}^n\right)_i - \sum_{j<i} a_{ij}\,x_j^n \right] . \tag{3.24}$$

We can express the i^{th} component of the vector $\Delta\boldsymbol{A}\,\boldsymbol{x}^n$ by

$$\left(\Delta\boldsymbol{A}\,\boldsymbol{x}^n\right)_i = -\sum_{j>i} a_{ij}\,x_j^n . \tag{3.25}$$

If we insert Eq. (3.25) in Eq. (3.24) we finally obtain the *Gauss-Seidel* iterative method

$$x_i^{n+1} = \frac{1}{a_{ii}} \left(b_i - \sum_{j>i} a_{ij}\,x_j^n - \sum_{j<i} a_{ij}\,x_j^n \right) , \tag{3.26}$$

for $i = 1, \cdots, m$.

If you look carefully at Eq. (3.20) and you compare to Eq. (3.26) you can see that the difference between both method is the used in *Gauss-Seidel* of updated values of x_j^n in the sum $\sum_{j<i} a_{ij}\, x_j^n$. In fact, in this sum x_j^n is *already* the value of x_j^{n+1} that we will have at the end of the "loop" $i = 1, \cdots, m$.

While trying to implement the *Jacobi* method with a computer program you might forgot that the x_i^n in Eq. (3.20) are the values at the previous iteration step and not the updated values. To do it right, you must store these values in an auxiliary array, otherwise your *Jacobi* implementation is wrong. The funny thing is that if you do this mistake [3] you end up with the more efficient *Gauss-Seidel* method. This is the only case that I can think of where you get better results through a programming mistake !

[3] I certainly did it !

Chapter 4

Ordinary differential equations

4.1 What is the problem ?

Chances are that you will very probably obtain an initial value problem when playing around with your favourite physical problem. Basically, we want to determine the function v of a variable $t \in \mathbb{R}$ verifying the following first order differential equation

$$\dot{v}(t) = f\left(v(t), t\right) \ , \tag{4.1}$$

where f is a function of the two variable v and t. To fully specify the problem, we need a "boundary condition". In this case, it amounts to give the value of $v(t_0)$, say v^0, the so-called *initial value*. The problem is to be solved for t greater than an "initial" parameter t_0 (taken to be 0 if not specified otherwise). The reason for quoting *initial* in the previous sentence is that even though we used t, obviously for a time evolution, the problem you might want to solve could well have nothing to do with a time evolution. Also note that since we have just one independent variable t, Eq. (4.1) is an *Ordinary Differential Equation*, or ODE for short.

Dealing only with the problem (4.1) might sound restrictive, and in some sense it is, but indeed it takes only a little algebra to convince oneself that indeed when solving problem (4.1) we are dealing with a much broader class of problems.

For instance, when we want to solve a good old problem of (classical) mechanics we need to solve a differential equation of order[1] 2. However, if we introduce a new variable, the momentum $\boldsymbol{p} = \frac{d\boldsymbol{r}}{dt}$, we obtain twice as much equations but of order 1. We can do this with the so-called *splitting*

[1] *order* appears in these lecture notes in many flavours, in this particular case it refers to the order of the highest derivative involded.

method. Let's consider the following example

$$\ddot{v}(t) = t\,\dot{v}(t) + v(t) \ . \tag{4.2}$$

We can introduce the new variable $u = \dot{v}$ to see that we obtain a system of first order equations

$$\begin{pmatrix} \dot{u} \\ \dot{v} \end{pmatrix} = \begin{pmatrix} t\,u + v \\ u \end{pmatrix} \ . \tag{4.3}$$

In fact, there is a price to the lowering of the equation order, we now have to deal with vectors, here $\boldsymbol{V} = \begin{pmatrix} u \\ v \end{pmatrix}$. The equation can now be written formally as $\dfrac{d\boldsymbol{V}}{dt} = \boldsymbol{F}(\boldsymbol{V}, t)$. We can even go further by introducing a new variable w equal to the independent variable t so that we obtain

$$\begin{pmatrix} \dot{u} \\ \dot{v} \\ \dot{w} \end{pmatrix} = \begin{pmatrix} w\,u + v \\ u \\ 1 \end{pmatrix} \ . \tag{4.4}$$

If we define $\boldsymbol{V} = \begin{pmatrix} u \\ v \\ w \end{pmatrix}$, now we have $\dfrac{d\boldsymbol{V}}{dt} = \boldsymbol{F}\left(\boldsymbol{V}\right)$ where t disappears from the equation which is then said to be *autonomous.*

In these lecture notes, we will only consider scalar problem for v. In some situations, when no loss in generality could be expected, we will also use autonomous equations to simplify the algebra. When studying the stability of numerical methods for the resolution of partial differential equations, again we will deal with scalars. Basically, the generalisation to vectors can be summarised by saying that we need to consider eigenvalues of matrices whenever we just consider scalar values. For the reader interested in gory details Henrici (1962) is definitely a good place to go.

There is a point that remains to be discussed : when can we be sure to have a unique solution to the problem depicted by Eq. (4.1) ? There is indeed a theorem due to *Lipschitz.* If f is continuous and defined on a domain in v and t (where we want to solve our problem) and if there exists a constant K such that

$$|f(v,t) - f(v',t)| \leq K\,|v - v'| \ , \tag{4.5}$$

for all v and v' on the considered domain, then there exists a unique solution to the initial value problem given in Eq. (4.1).

The fact that the solution exists does not mean that we can express it in a closed form, like a convergent series or an elliptic integral or whatsoever (this would mean - loosely speaking - that the problem is *integrable*). Take for example the 3-body problem for which more then a century ago *Bruns* and *Poincaré* showed that such a closed form cannot be guaranteed[2]. So even in such well-posed cases you might need to fire up that good old `emacs` editor and start coding something, if you want to get at least some insights on the solution...

There is a lot of good references on the numerical integration of ODEs. If the reader wants to go further here comes a few. We strongly advise LeVeque (2007). Most of the material presented here follows more or less the presentation of this author, in particular concerning the link between ODEs and PDEs. If you want to read just one, read this one ! Ralston & Rabinowitz (2001) is also a good place to go with an older exposition but still worth reading and more or less coherent with LeVeque (2007). If you want more mathematical details you can either have look at Trefethen (1996) [3] or for the more adventurous at Henrici (1962) where you can find the details with all the mathematical rigor. When time has come, you might want to implement all theses idea in the form of a computer program. Press et al. (1986) - "The numerical recipies" - will then be very useful.

4.2 Euler methods

Before entering the details of the numerical integration of ODEs, it is very instructive to study the simplest method due to *Euler*. This method is the member of a more general class, the *one-step methods*, for which we advance our approximate solution from one "time" step to the next. We thus assume that we want to obtain an approximate solution of Eq. (4.1) for discrete values of t, i.e. for $t_n = t_0 + n\,k$ where k is a constant called the *integration step* and n is a positive integer. Let's call these approximations u^n. We use a superscript for indexing this value for reasons that will be clearer in the next chapter concerning partial differential equations. We change the notation from v to u at this point because we do not want to rise the confusion between this approximation and $v(t_n)$, also noted v^n, the value of the *exact* solution. While devising a numerical method for integrating the ODE we might want to ensure $u^n \approx v(t_n)$ but of course we do not have in general the equality $u^n = v(t_n)$.

Let's assume that we have an approximation u^n for $t = t_n$. In order to estimate u^{n+1} we can approximate the function $u(t)$ by a straight line. Since we have $\dot{v}(t) = f(v(t), t)$ it is natural to estimate the slope of this straight line by $f(u^n, t_n)$. Now, using this value we obtain for $v(t) \in (t_n, t_{n+1})$ the following approximation

$$v(t) \approx f(u^n, t_n)\ (t - t_n) + u^n\ . \tag{4.6}$$

[2]and it is really unlikely to exist, but this is really a long story related to the notion of *chaos*...

[3]available at `http://people.maths.ox.ac.uk/trefethen/pdetext.html`.

Considering the above equation for $t = t_{n+1}$ gives us the Euler method

$$u^{n+1} = u^n + f(u^n, t_n)\, k \ , \tag{4.7}$$

since $k = t_{n+1} - t_n$, which is a recurence relation between the u^n that must be started by using the initial condition $u^0 = v^0$.

By re-writting Eq. (4.7) in the following manner

$$\frac{u^{n+1} - u^n}{k} = f(u^n, t_n) \ , \tag{4.8}$$

we can see it with a different point of view. We can say that $\frac{u^{n+1}-u^n}{k}$ is the discrete version of the continuous operator $\frac{d}{dt}$ where we replace it by an expression involving finite differences.

We have seen previously in these lecture notes, that we can use the backward finite difference for estimating the derivative of a function. In this case we obtain

$$\frac{u^{n+1} - u^n}{k} = f(u^{n+1}, t_n) \ , \tag{4.9}$$

where we used u^{n+1} on the right-hand side. By doing so, we must face a supplementary problem because we cannot obtain u^{n+1} directly we need some more work, especially if f is a non-linear function of v. For an arbitrary f it can be necessary to use iterative methods to solve the non-linear equation for u^{n+1} (e.g. Newton-Raphson method). These kind of method, i.e. when we cannot get u^{n+1} directly, is called an *implicit* method while whenever we can (as in Eq. 4.8) it is called *explicit*.

At this stage, we can think that it is better to avoid *implicit* methods because they will be with no doubt more difficult to implement. The awful truth is that in some instances implicit methods are the best choice because they are *stable* while explicit methods can be *unstable* in some cases. We postpone the discussion of stability at the end of this chapter, but an unstable numerical method will lead to u^n values diverging exponentially from the exact solution in some unfortunate circumstances : when the step k is too large for instance. We will learn to determine the upper limit on k for a method to be stable, but that means we cannot integrate as fast as we want since our steps cannot be larger than a given value. Nevertheless, we will learn as well to not always expect miracles from implicit schemes[4].

As a side effect of considering the Euler's method, let's note that if we take the arithmetic mean of Eq. (4.8) and Eq. (4.9) we obtain another method, the so-called *trapezoidal rule*,

$$\frac{u^{n+1} - u^n}{k} = \frac{1}{2}\,\big(f(u^{n+1}, t_n) + f(u^n, t_n)\big) \ , \tag{4.10}$$

[4]unless you are dealing with stiff equations, a topic which for the moment is not included in these lectures.

which is again implicit. Indeed, if f depends only on t we recover the well known extended *Newton-Cotes* formula of the same name. It is also called the *Crank-Nicolson* method in the framework of the numerical solutions of partial differential equations (see next chapter).

4.3 Truncation errors and consistency

In the *Euler* methods we approximate the function by a straight line and by doing so we make a truncation error that we want to estimate. Let's assume that we have solved the problem exactly at $t = t_n$, in other words we have $u^n = v(t_n)$. This assumption may sound strange, because in a real computation it is never true (unless the starting step is the initial condition at which we impose $u^0 = v(t_0)$). But we make this hypotesis because our present goal is to study just what happens in one step, i.e. how big is the difference between the exact and the numerical solution after just one step when starting from exactly the same values.

We then define the one-step error as the difference between the numerical and the exact solution from t_n to t_{n+1}:

$$e^{n+1} = u^{n+1} - v(t_{n+1}) \ , \tag{4.11}$$

under the assumption that $u^n = v(t_n)$. In the case of the explicit *Euler* method Eq. (4.8) it gives

$$e^{n+1} \ = \ u^n + k\, f(u^n, t_n) - \left(v(t_n) + \dot{v}(t_n)\, k + \frac{k^2}{2}\, \ddot{v}(t_n) + O(k^3) \right)$$

and since $u^n = v(t_n)$ and consequently $f(u^n, t_n) = f(v(t_n), t_n) = \dot{v}(t_n)$, we obtain finally

$$e^{n+1} = -\frac{k^2}{2}\, \ddot{v}(t_n) + O(k^3) \ . \tag{4.12}$$

We see that the leading term in e^{n+1} is a $O(k^2)$. Generally, a method is said to be of order p if $e^{n+1} = O(k^{p+1})$, so that the *Euler* method is order 1.

Why k^{p+1} and not k^p ? Because this error can be used as a rough estimation for the error made in a fixed time interval $T = Nk$, where N is the total number of steps we make on the whole. To estimate this total error we sum up the one-step error N times, where for a fixed T, $N \propto \frac{1}{k}$ and the total error can be estimated as $\frac{1}{k}\, O(k^{p+1}) = O(k^p)$. When we say that the scheme is of order p we mean then that in a fixed time interval, if we divide the time step k by 2, the total error is roughly divided by 2^p.

We can alternatively write things in the following way

$$e^{n+1} = k\, f(u^n, t_n) + u^n - v(t_{n+1}) \ ,$$

and

$$e^{n+1} = -k \left[\frac{v(t_{n+1}) - v(t_n)}{k} - f\left(v(t_n), t_n\right) \right] \,, \tag{4.13}$$

using again the fact that $u^n = v(t_n)$ and $f(u^n, t_n) = f(v(t_n), t_n)$.

Let the term within bracket in the rhs of the above equation be noted τ^{n+1}. It looks like, *but is not*, the following expression

$$\frac{u^{n+1} - u^n}{k} - f(u^n, t_n) \,, \tag{4.14}$$

which cancels out because the u^n are the solutions to this algebraic equation.

Now, if we inject the actual solution v to the ODE in Eq. (4.14) we obtain the bracketed term in Eq. (4.13), but we do not get 0 this time because the actual solution v does not verify our finite differencing scheme. The value that we get is indeed τ^{n+1}. This error is also caused by our discrete representation of a continuous problem and is therefore a truncation error in the same way e^{n+1} is. The jargon is here to make things more obscure[5] than they actually are because τ^{n+1} is frequently called the *truncation error* as opposed to the *one-step* error, e^{n+1}. Unfortunately, both errors belongs to the class of "truncation errors" obtained whenever one treats a continuous problem in a discrete way.

While it is more natural - at least we think so - to introduce the one-step error, it is indeed easier to work with the truncation error because it usually simplifies the algebra. In the case of the *Euler* method we can see that both errors are related simply by $e^{n+1} = -k\,\tau^{n+1}$. Another supplementary formal advantage of τ^{n+1} respect to e^{n+1} is the the fact that if the method is order p we have simply $\tau^{n+1} = O(k^p)$.

Conceptually, while e^{n+1} is simply an error on the solution, it is more difficult to get a picture of what τ^{n+1} indeed is. Remember its definition for our simplest *Euler* method:

$$\tau^{n+1} = \frac{v(t_{n+1}) - v(t_n)}{k} - f\left(v(t_n), t_n\right). \tag{4.15}$$

In fact we might want to compute the truncation error in the equation itself, whenever we inject our numerical approximations u^n in the actual ODE Eq. (4.1). However, we cannot do that because the u^n are discrete values and it makes no sense to talk about inserting them in the differential operator inside the equation (4.1). We then define a truncation error τ^{n+1} by proceeding the other way around: we inject the solution v in our numerical scheme. It has the flavour of "To which level of accuracy does my approximation verifies the original equation ?", but *is not*! Instead, it is rather an answer to the question "to which level of accuracy does the

[5]Well, as usual . . .

exact solution verifies my approximation to the equation". If you do not feel at ease with the definition of τ^{n+1} remember that it is simply related to e^{n+1} by $e^{n+1} = -k\,\tau^{n+1}$ as derived above. Indeed it is not that obvious: when considering more complicated schemes, this relation between e^{n+1} and τ^{n+1} is true only for the leading terms in k, but this will be enough for any practical purposes.

The fact that $\tau^{n+1} = O(k^p)$ implies that $\tau \to 0$ as $k \to 0$. This property is called the *consistency* of the numerical method, and because $e^{n+1} \sim -k\,\tau^{n+1}$ it implies that $e^{n+1} \sim k^2$ at least. If we make the approximation that after N steps the global error just adds at each step[6], then the error after a time $T = Nk$ will actually go to 0 for a consistent scheme. This property is really the minimum we can require to a numerical integration. It would in fact sound strange if some truncation error introduced by the numerical discretization actually did not decrease by lowering the time step!

We will now look to what happens to the one-step and truncation errors in the case of the implicit Euler method. Starting from the definition put in the form (4.13) we have

$$e^{n+1} = -k\left[\frac{v(t_{n+1}) - v(t_n)}{k} - f\left(u^{n+1}, t_{n+1}\right)\right] , \tag{4.16}$$

where u^{n+1} must *not* be replaced by v^{n+1}! A little more work is in fact needed in order to make the link between e^{n+1} and τ^{n+1}. Since we have

$$f(u^{n+1}, t_{n+1}) = f(v(t_{n+1}), t_{n+1}) + \partial_v f(v(t_{n+1}), t_{n+1})\, e^{n+1} + O\left((e^{n+1})^2\right) , \tag{4.17}$$

we have therefore

$$\begin{aligned} e^{n+1} = & - k\left[\frac{v(t_{n+1}) - v(t_n)}{k} - f\left(v(t_{n+1}), t_{n+1}\right)\right] \\ & - k\,\partial_v f(v(t_{n+1}), t_{n+1})\, e^{n+1} + k\,O\left(\left(e^{n+1}\right)^2\right) , \end{aligned} \tag{4.18}$$

and finally we again have the formal relation

$$e^{n+1} = -k\left[\frac{v(t_{n+1}) - v(t_n)}{k} - f\left(v(t_{n+1}), t_{n+1}\right)\right] + k\,O\left(e^{n+1}\right) \tag{4.19}$$

which after Taylor expansion turns out to be $O(k^2)$ as for the explicit scheme. Because the bracketed term in (4.19) is by definition τ^{n+1}, we see that to leading terms in powers of k we again have $e^{n+1} \approx -k\,\tau^{n+1}$.

When one has to verify the order of the method or just check the consistency, the truncation error τ is more easy to work with even if the one-step error is immediately meaningful.

[6]this is true only asymptotically for $k \to 0$, but since in this limit on a finite time we add an infinite number of errors, some care is needed : this is the problem of *stability* as addressed in the next Section

For explicit (or even implicit) Euler the scheme looks evidently consistent, but this is not so for more complicated method like trapezoidal or high-order Runge-Kutta or multistep methods. In general one can write extremely complicated schemes, but consistency is really the 0-th order requirement for an algorithm to be useful and not to be doomed to the trash bin. Just to give general consideration, let's imagine that a generic one-step method for the equation (4.1) $\dot{v}(t) = f\left(v(t), t\right)$ reads as

$$u^{n+1} = u^n + k\Psi(u^n, u^{n+1}, t_n, k) \ , \tag{4.20}$$

(Ψ depending evidently on f). By definition the truncation error is

$$\tau^{n+1} = \frac{v(t_{n+1}) - v(t_n)}{k} - \Psi\left(v^n, v^{n+1}, t_n, k\right) \ . \tag{4.21}$$

If we Taylor-expand $\Psi(v^n, v^{n+1}, t_n, k)$ about $\Psi(v^n, v^n, t_n, 0)$ we get:

$$\begin{aligned}\Psi\left(v^n, v^{n+1}, t_n, k\right) &= \Psi\left(v^n, v^n + k\dot{v}^n + O(k^2), t_n, k\right) \\ &= \Psi\left(v^n, v^n, t_n, 0\right) \\ &\quad + k\dot{v}^n \Psi_{v^{n+1}}\left(v^n, v^n, t_n, 0\right) \\ &\quad + k\Psi_k\left(v^n, v^n, t_n, 0\right) + O(k^2)\end{aligned}$$

where subscripts in Ψ indicate derivatives with respect to the indicated variable. Taking $k \to 0$, to satisfy consistency one needs $\Psi(v^n, v^n, 0, t_n, t_n) = f(v^n, t_n)$. The reader can check that this holds e.g. for the trapezoidal scheme.

4.4 Convergence

What could we ask from a numerical method whose purpose is to approximate the solution $v(t)$ of and ODE ? As we saw previously, we hope to obtain such an approximation by using a *discrete* ersatz of the original ODE. The quality of our approximation depends on the integration step k. We consider only consistent methods meaning that our ersatz equation transforms into the continuous ODE, at least locally: we hope to obtain better approximation when $k \to 0$. However, we need more than that. For a fixed time $t_N = T = N\,k$, we demand that our approximation converge to the true solution for an infinite number of steps, or in other words $\lim_{N\to\infty} u^N = v(T)$. Of course, consistency is a necessary condition, but is it sufficient ? Luckily, for one-step methods it is.

We start by considering the explicit *Euler* method and then give a proof of convergence for any one-step method.

4.4.1 Qualitative behaviour of the numerical error

First, let's have a naive and qualitative look to the behaviour of the error in the *Euler* method. We have two contributions to take into accounts. We have the truncation error arising from the approximation of the function by straight-lines. We also have the rounding errors proportional to the rounding unit u. Assuming that we start to integrate at $t_0 = 0$ then for N steps in total we have $k = \dfrac{T}{N}$ for a fixed time T. The total (truncation+round-off) error that we have is then given by

$$e(k) \approx \frac{T}{k}\, e + \frac{T}{k}\, u \ , \tag{4.22}$$

with e is $e = \max\big\{e^{n+1} | n = 1, \cdots, N\big\}$ and where we assume that the rounding error is $\approx u$.

For the *Euler* method we saw that $e^{n+1} \propto k^2$ since the method is order 1 so that the behaviour of the total error is expected to be

$$e(k) \approx \alpha\, k + \beta\, \frac{u}{k} \ , \tag{4.23}$$

where α and β are constants that we assume close to unity [7].

Eq. (4.23) is definitely not perfect. The careful reader might have identify that something is missing here which is the purpose of the next section. But still, we can see the combined effect of truncation and round-off through Eq. (4.23). When $k \to 0$, the truncation error goes down but since we are doing more floating point operations the rounding error goes up. This competition between both effects as the consequence that we cannot reduce k as far as we want to increase the precision. There is a lower limit for which, by reducing k, we would obtain worse results, a situation that seems somehow counter-intuitive. By setting the derivative $e'(k) = \alpha - \beta\, \frac{u}{k^2}$ to zero we see that the optimal value for k is $\sqrt{\frac{\beta}{\alpha}}\, u^{\frac{1}{2}}$. By inserting this value in Eq. (4.23) we obtain the results that the lowest value for the total error is proportional to $u^{\frac{1}{2}}$.

For double precision floating point numbers (`double`) this still gives us a reasonable lower limit scaling as $\sim 10^{-8}$ relatively. However, for single precision (`float`) this lower limit $\sim 10^{-4}$ is clearly not sufficient. So once again, the numerical computation must be carried on with `double` floating point numbers or we might get into troubles.

If the numerical method is order p we must modify slightly Eq. (4.23) in the following manner

$$e(k) \approx \alpha\, k^p + \beta\, \frac{u}{k} \ . \tag{4.24}$$

In this case, the upper limit for the total error scales as $u^{\frac{p}{p+1}}$ and we clearly see the advantages of higher order method in that case.

[7] Of course, this is ok only for our qualitative approach.

4.4.2 What really matters: the *global* error

In Eq. (4.23) and (4.24) things are simple, but in reality they are *less* simple. As we already said the one-step error e^{n+1} assumes that for $t = t_n$ we have $u^n = v(t_n)$, but if one needs to measure the true error during a whole numerical integration this assumption cannot be made. Indeed, during an integration step we are generating an error on something that was previously containing numerical errors. If we are lucky enough this process will not lead to the "explosion" of our numerical solution but if we are not, errors could be amplified and we might get into troubles. Let's look at this in detail.

Starting from the initial condition $u^0 = v(t_0)$ we define the actual error (*global error*) after the step $n \to n+1$ on $v(t_{n+1}) \equiv v^{n+1}$ by

$$E^{n+1} = u^{n+1} - v^{n+1} \ , \tag{4.25}$$

which must not be confused with the one-step error e^{n+1} because we do not assume here that the problem is exactly solved for the previous time step: $u^n \neq v(t_n)$ *a priori*. E^{n+1} takes into account all the errors accumulated step after step.

For a numerical integration method to be really useful we must demand that the error E^n goes to zero when the time step k goes to zero, and this on a fixed finite time T (which implies that the number of time steps $N = T/k$ goes to ∞). This is the natural definition of the *convergence* for the *Euler* method.

If in this limit we can find an upper bound to $||E^{n+1}||$ that can be made arbitrarily small then our job is done and we have proved the convergence of the *Euler method* (just to increase generality, we study the convergence in some generic norm $||\cdot||$ to not limit ourselves to a scalar equation, for which a simple modulus $|\cdot|$ would be enough). We will find that the property of convergence is related to the fact that the truncation error τ goes to zero as $k \to 0$ (i.e. to the *consistency* of the scheme) and to some form of *stability* (which in this case looks trivial but it will be less trivial for multi-step schemes, see **??**, and far less trivial when switching from ODEs to PDEs).

Recalling the expression of the *Euler* method Eq. (4.7) and the definition of the truncation error Eq. (4.15) that can be re-written like this

$$v^{n+1} = v^n + k\, f(v^n, t_n) + k\, \tau^{n+1} \ , \tag{4.26}$$

we obtain for the global error the following expression

$$\begin{aligned} E^{n+1} \ &= \ u^n + k\, f(u^n, t_n) \\ &- \ \left[v^n + k\, f(v^n, t_n) + k\, \tau^{n+1}\right] \ . \end{aligned} \tag{4.27}$$

Recognizing $u^n - v^n$ as E^n and regrouping terms we finally have

$$E^{n+1} = E^n - k\tau^{n+1} + k(f(u^n, t_n) - f(v^n, t_n)) \tag{4.28}$$

Taking norms and applying the triangular inequality:

$$\|E^{n+1}\| \leq \|E^n\| + k\|\tau^{n+1}\| + k\|f(u^n, t_n) - f(v^n, t_n)\| \tag{4.29}$$

Now from the *Lipschitz* condition there exists some positive constant K such that :

$$\|f(v^n, t_n) - f(u^n, t_n)\| \leq K\|u^n - v^n\|$$

which since $1 + kK = |1 + kK|$ implies

$$\|E^{n+1}\| \leq |1 + kK|\, \|E^n\| + k\|\tau^{n+1}\| \tag{4.30}$$

The global error is then bounded by the sum of two quantities: one is the truncation error, the other is the amplification of the global error at the previous step.

For a given value of k we have an upper limit, $\tau(k)$, on the values of $\|\tau^n\|$ for $n = 0, \cdots, N$, N being the total number of steps (i.e. $T = k\,N$). We can take $\tau(k) = \max_{n=0,\cdots,N} \{\|\tau^n\|\}$ for instance, recalling that this upper limit is itself a function of k, a $O\,(k)$ in the particular case of the *Euler* method.

We have then

$$\|E^{n+1}\| \leq |1 + kK|\|E^n\| + k\,\tau(k) \tag{4.31}$$

Then by induction one has :

$$\|E^n\| \leq |1 + kK|^n\|E^0\| + k\sum_{m=1}^{n} |1 + kK|^{n-m}\,\tau(k) \tag{4.32}$$

Now because we are looking for a bound in the limit $k \to 0$, $N \to \infty$ ($T = Nk$ fixed), it is enough that for $0 \leq m \leq n$, $|1 + kK|^{n-m}$ be uniformly bounded by some constant in the given time interval. If on the contrary $|1 + kK|^n \to \infty$ in such a limit, no bound on the global error would be obtained this way. What is then necessary is some form of *stability*, which ensures that the initial global error E_0 (which is actually strictly equal to zero without rounding errors) and the accumulated truncation error does not get amplified without bounds when lowering the time step and increasing the number of steps. In this case it is easy to satisfy such a requirement, because since $|1 + kK| \geq 1$ one has for $0 \leq m \leq n$:

$$|1 + kK|^{n-m} \leq |1 + kK|^n \leq e^{kKn} = e^{Kt_n}$$

where we introduced the total integration time from 0 to n as $t_n = n\,k$. When dealing with multi-step methods or with PDEs the requirement needed is in fact much less trivial. Going further one has

$$\begin{aligned} \|E^n\| &\leq e^{Kt_n}\|E^0\| + ke^{Kt_n}\sum_{m=1}^{n} \tau(k) \qquad (4.33)\\ &= e^{Kt_n}\|E^0\| + kne^{Kt_n}\,\tau(k) \\ &= e^{Kt_n}(\|E^0\| + t_n\,\tau(k)). \end{aligned}$$

Since the truncation error $\tau(k) \to 0$ as $k \to 0$, we found that for a fixed finite integration time T, $\|E^N\| \to 0$ as $k \to 0$, $N \to \infty$ so the method is convergent provided that $E^0 = 0$ i.e. we ignore the effect of rounding error for the initial condition.

If one integrates the usual differential equation (4.1) with a generic one-step method in the form (4.20) as

$$u^{n+1} = u^n + k\Psi(u^n, u^{n+1}, t_n, k) \ ,$$

convergence can be proved if some supplementary hypoteses on the function Ψ (which is a functional of f) are satisfied.

For an explicit scheme in which $\Psi = \Psi(u^n, t_n, k)$ only we can ask for Ψ *Lipschitz countinuous* in u, at least when k is restricted to a suitable neighborhood of 0 and for any t in the considered interval, which accounts in fact to ask

$$|\Psi(u, t, k) - \Psi(v, t, k)| \le \bar{K}(k)|u - v|$$

with $0 \le \bar{K}(k) \le K =$ const for k in a neighborhood of 0. This is enough because we are interested in the limit $k \to 0$. In this case the demonstration of convergence follows exacly the same lines as for the explicit Euler method.

For a function f linear in v of the kind $f(v, t) = \lambda v + g(t)$ (λ independent of time), in which Ψ itself is linear in each argument, convergence can be worked out explicitely even in the implicit case by realizing that in most cases the scheme can be written as

$$u^{n+1} = u^n + k\psi(k)u^n + \gamma(t_n, t_{n+1}) \tag{4.34}$$

i.e. with a functional which is simply a multiplicative operator $\psi(k)$ (sufficiently differentiable) and the temporal dependence remains a source term. For example the reader can check that for the trapezoidal scheme in the scalar case,

$$\begin{aligned} \psi(k) &= \frac{\lambda}{1 - \lambda k/2} \\ \gamma(t_n, t_{n+1}) &= \frac{k}{2(1 - \lambda k/2)}(g(t_n) + g(t_n + 1)) \end{aligned} \tag{4.35}$$

while for any Runge-Kutta the form (4.34) holds with a polynomial ψ. We proceed to show how convergence requirement looks like in this simplified case. Starting from a reformulation of the truncation error one has

$$v^{n+1} = v^n + k\,\psi(k)v^n + \gamma(t_n, t_{n+1}) + k\,\tau^{n+1} \ ,$$

then the global error can be written as

$$\begin{aligned} E^{n+1} &= u^n + k\,\psi(k)u^n + \gamma(t_n, t_{n+1}) - \left[v^n + k\,\psi(k)v^n + \gamma(t_n, t_{n+1}) + k\,\tau^{n+1}\right] \\ &= (u^n - v^n) - k\tau^{n+1} + k\,\psi(k)(u^n - v^n) \\ &= (1 + k\psi(k))\,E^n - k\tau^{n+1} \end{aligned}$$

and proceeding the same way as before one gets

$$\|E^n\| \leq |1 + k\psi(k)|^n \|E^0\| + k \sum_{m=1}^{n} |1 + k\psi(k)|^{n-m} \tau(k). \tag{4.36}$$

Convergence then holds if in the given time interval $[0, T]$, $|1 + k\psi(k)|^{n-m}$ is uniformly bounded by some constant (which may depend on the total integration time T), at least for k in a suitable neighborhood of 0. If

$$|1 + k\psi(k)|^{n-m} \leq C(T)\ , m = 0 \ldots N$$

we can write

$$\|E^n\| \leq C(T)\|E^0\| + TC(T)\tau(k). \tag{4.37}$$

For the Euler explicit scheme, we obtained in fact $C(T) = e^{KT}$.

Boundedness of $|1 + k\psi(k)|^{n-m}$ is verified if $\psi(k)$ is bounded as $k \to 0$, because in this case $|1 + k\psi(k)|$ can be bounded by a linear function of kind $|1 + kK|$ and we recover the explicit Euler case. But for consistency $\psi(k)$ must tend to λ as $k \to 0$ (because the truncation error must tend to zero!), so at least in this simple case convergence is ensured.

What we learned so far? That under assumptions that are quite reasonable for common methods, consistent schemes are convergent since from (4.37) the global error until a time T is bounded by a function $C(T)$ times a term $\tau(k)$ which is at least $O(k)$ and can go to 0 as $k \to 0$ at a finite time (one nevertheless discards the error at time 0 that depends on the floating-point approximation on the initial condition). There is then some sort of "stability" in amplifying the errors. This concept (referred as *zero-stability* because relevant in the limit $k \to 0$) will be more rigorously defined in the case of multi-step methods (one-step methods being trivially zero-stable), where we will learn that zero-stability + consistency imply convergence.

4.5 More on one-step methods

Up to now, to get in touch with numerical integration methods we played around with *Euler* methods which are members of a more general class that we already mentioned the so-called *one-step* methods. The framework is simple: we advance the method from time t_n to t_{n+1} with a given algorithm.

We already saw three examples: backward/forward *Euler* and the trapezoidal rule. Well, basically if we restrict ourselves only to the use of u^n, $f(u^n, t_n)$ and $f(u^{n+1}, t_{n+1})$ as our available [8] information for determining u^{n+1} we cannot do much better that recover these three old friends. Why ? Let's have a look to what the method of undetermined coefficients has to say about this.

[8]If the method is implicit, i.e. involving $f(u^{n+1}, t_{n+1})$, some extra work is of course needed.

Regarding the information that we have we might wan to try a method of the following type

$$u^{n+1} = a\,u^n + k\,\left[b\,f(u^n, t_n) + c\,f(u^{n+1}, t_{n+1})\right]\ , \tag{4.38}$$

where a, b and c are unknowns. To determine these coefficients we proceed as usual by inserting polynomials of successive degree for $v(t)$ in Eq. (4.38) until we have 3 (linear) equations. If we chose to insert 1, $t-t_n$ and $(t-t_n)^2$ we obtain the three following equations

$$\begin{cases} a & = & 1 \\ b+c & = & 1 \\ 2\,b & = & 1 \end{cases}\ , \tag{4.39}$$

whose solution is obvious.

We could have inserted only the first two polynomials. In that case, we use all the information but we simply do not insure the largest order for the method. This is of course unwise, but let's write down in this case the numerical methods we obtain

$$u^{n+1} = u^n + k\,\left[\alpha\,f(u^n, t_n) + (1-\alpha)\,f(u^{n+1}, t_{n+1})\right]\ , \tag{4.40}$$

for $\alpha \in \mathbb{R}$.

This set of methods clearly includes the *Eulers*' as expected. For $\alpha \neq \frac{1}{2}$, the method is only first order [9], so we better used *Euler* methods anyway. The cost of evaluating f two times for gaining nothing is not the best idea we could have. For $\alpha = \frac{1}{2}$ we recover the trapezoidal rule.

Basically, that is it. We cannot do better than order 2 with the provided information. However, it is legal [10]-indeed a good idea-to generate *more* information. For instance, we could

1. try and evaluate f at carefully selected times $t \in [t_n, t_{n+1}]$ and mixed this up to cook up a higher order method. Doing so, we obtain the *Runge-Kutta* method class.

2. f being the derivative of v, we could compute derivative of v of successive orders and add it to our recipe and obtain the *Taylor series* method.

4.5.1 Taylor series methods

Let's consider $v(t_{n+1})$ and develop it around t_n, we obtain

$$\begin{aligned} v(t_{n+1}) &= v(t_n) + k\,\dot{v}(t_n) + \frac{k^2}{2}\,\ddot{v}(t_n) + O\left(k^3\right)\ , \\ &= v^n + k\,f(v^n, t_n) + \frac{k^2}{2}\,\ddot{v}(t_n) + O\left(k^3\right)\ . \end{aligned} \tag{4.41}$$

[9]The method of undetermined coefficient already gives us this information but if you are not convince it is time for you to play with *Taylor* expansions.

[10]At least, in Europe at the time of this writing ...

This could be our source of inspiration for our numerical method, using the following algorithm

$$u^{n+1} = u^n + k\, f(u^n, t_n) \ , \tag{4.42}$$

directly inspired from Eq. (4.41). Eq. (4.42) is the forward *Euler* method.

What is the error we make by using Eq. (4.42) ? If we are interested in the one-step error we just have to substract Eq. (4.41) from Eq. (4.42) to obtain

$$e^{n+1} = -\frac{k^2}{2}\, \ddot{v}(t_n) + O\left(k^3\right) \ , \tag{4.43}$$

remembering that we assume the problem is solved exactly at $t = t_n$, i.e. $v^n = u^n$.

So basically we *Taylor* expand our solution around $t = t_n$ up to the desired order, say p, and the truncation error is $O\left(k^{p+1}\right)$, i.e. the method is order p. So this is an easy way to obtain a higher order method.

Let's generalise the process. We have

$$v(t_{n+1}) = \sum_{l=0}^{p} \frac{1}{l!}\, v^{(l)}(t_n)\, k^l + O\left(k^{p+1}\right) \ . \tag{4.44}$$

We must now remember that $\dot{v}(t) = f\left(v(t), t\right)$. Consequently, we have also $\ddot{v}(t) = f\,\frac{\partial f}{\partial v} + \frac{\partial f}{\partial t}$ and in general we have

$$v^{(l+1)}(t) = \left(f\,\frac{\partial}{\partial v} + \frac{\partial}{\partial t}\right)^l f \ . \tag{4.45}$$

Eq. (4.44) together with Eq. (4.45) suggests the following order p method

$$u^{n+1} = u^n + \sum_{l=0}^{p} \frac{1}{(l+1)!} \left(f\,\frac{\partial}{\partial v} + \frac{\partial}{\partial t}\right)^l f(u^n, t_n)\, k^{l+1} \ . \tag{4.46}$$

With the above method we can generate a one-step method with the order of our choice. However, there are a few drawback to this procedure. First, we must evaluate numerically the values of the derivatives of f and this can be a costly computation. Second, we do not have necessarily the expressions of these derivative. For instance, f can be the results of a computer program that cannot be expressed in term of usual mathematical functions. In that case we would need to estimate the derivatives numerically, introducing a new source of error. Third, The method depends explicitly on f: change f and the numerical method is different. We do not want this property because usually we will call a `C` (or whatever) function from a library to solve our equation with an arbitrary f. With the method depicted in Eq. (4.46) that would mean to either being able to do *formal* derivation or at least estimate them numerically.

Perhaps we could do better by using the values of $f(v(t), t)$ evaluated for some selected times t and recover the terms in the development of Eq. (4.46) up to the desired order. This leads us to the next section about *Runge-Kutta* methods.

4.5.2 Runge-Kutta methods

Runge Kutta (RK) methods fall in the category of explicit one-step methods, and they can be considered a "natural" extension of the simple explicit *Euler*. Before we start, let's take an example to motivate the RK methods.

With *Euler* methods, either explicit or implicit, we use the value of the derivative at t_n or t_{n+1} which gives us an order 1 method. It is too bad we do not have access to the derivative just in the middle of this time interval because with this centered difference we could hope to obtain a second order method. We do not have this value but perhaps we could make a guess. First, using forward *Euler* we estimate the value of v in the middle of the interval. Let's call this estimation $\tilde{u}$, we thus have

$$\tilde{u} = u^n + \frac{k}{2} f(u^n, t_n) \ . \tag{4.47}$$

Then, we simply use the $\tilde{u}$ value to do our step like this

$$u^{n+1} = u^n + k\, f\left(\tilde{u}, t_n + \frac{k}{2}\right) \ . \tag{4.48}$$

We are not trough yet because we need to check that the method described by Eq. (4.47) and (4.48) gives a consistent method and with an order at least 2, gaining something with respect to *Euler* methods.

To simplify the algebra, let's assume that the ODE is autonomous (i.e. f does not depend on t). The truncation error is then

$$\tau^{n+1} = \underbrace{\frac{v(t_{n+1}) - v(t_n)}{k}}_{①} - \underbrace{f\left(v(t_n) + \frac{1}{2}\, k\, f(v(t_n))\right)}_{②} \ . \tag{4.49}$$

The term ① is easy to deal with, we have

$$① = v'(t_n) + \frac{k}{2} v''(t_n) + \frac{k^2}{6} v'''(t_n) + O\left(k^3\right) \ . \tag{4.50}$$

For term ② we have

$$\begin{aligned} ② = f\left(v(t_n)\right) &+ \frac{k}{2} f\left(v(t_n)\right) f_v\left(v(t_n)\right) \\ &+ \frac{k^2}{8} f^2\left(v(t_n)\right) f_{vv}\left(v(t_n)\right) + O\left(k^3\right) \ , \end{aligned} \tag{4.51}$$

where the partial derivative of f respect to v is noted f_v.

Now, remembering that we have $v''(t) = f\left((v(t)\right)\, f_v\left(v(t)\right)$ and substrating ① from ② we obtain finally

$$\begin{aligned} \tau^{n+1} &= \left[\frac{1}{6}\, v'''(t_n) - \frac{1}{8}\, f^2\, f_{vv}\right]\, k^2 \\ &= \frac{1}{24}\, \left[f^2\, f_{vv} + 4\, f\, f_v^2\right]\, k^2 + O\left(k^3\right)\ , \end{aligned} \tag{4.52}$$

where we omitted the dependence of f for clarity and where we used the relation $v'''(t_n) = f\, f_v^2 + f^2\, f_{vv}$.

We thus have $\tau^{n+1} = O\left(k^2\right)$ and obtained an order two one-step method! The price to pay is modest. We must proceed in several stages (here two) and evaluate f many times but of course we expected that: *no pain, no gain* [11].

Now, it is time to generalise the procedure. Let's consider again the usual equation (4.1) $\dot{v}(t) = f(v(t), t)$. Explicit *Euler* consists in estimating the value of the function u^{n+1} at time $t_{n+1} = t_n + k$ by adding to the value u^n the quantity $kf(u^n, t_n)$, i.e. the linear increment obtained by replacing the true function by the straight line built with the value of the derivative at (u^n, t_n).

RK methods are more sophisticated in that they use successive estimates of the derivative at intermediate times $t_i \in [t_n, t_{n+1}]$ to build a final approximation for u^{n+1} by linearly combining these estimates.

Proceeding as in the above order 2 example, we start first from the usual (*Euler*) approximation of the increment $u^{n+1} - u^n$, which we call J_1 :

$$J_1 = kf(u^n, t_n).$$

We then consider the derivative $f(v(t), t)$ at some intermediate time $t = t_n + c_2 k \leq t_n + k$ $(c_2 \leq 1)$ by taking some estimation for the value of v at such time. We use the previously calculated J_1 to estimate such value, i.e. $v(t_n + c_2 k) \approx u^n + c_2 J_1$ from which $f(v(t), t) \approx f(u^n + c_2 J_1, t_n + c_2 k)$.

We now use this other estimation for the derivative to calculate another approximation for the increment like this,

$$J_2 = kf(u^n + c_2 J_1, t_n + c_2 k)\ .$$

So we ended up with two possible increments J_1 and J_2, and we can use both to calculate some estimation for $v(t)$ at some other intermediate time $t = t_n + c_3 k$. We use J_1 for a fraction of interval a_{31} and J_2 for another fraction a_{32}, in such a way that $c_3 = a_{31} + a_{32}$ (a_{31} and a_{32} are not necessarily

[11] Ages ago, I was in a basketball summer camp and we had this on our tees together with *go hard or go home* :-) But hey, this is not a strict recommendation: we're human beings not machines ...

positive but for their sum $0 \leq c_3 \leq 1$). So $v(t_n + c_3 k) \approx u^n + a_{31}J_1 + a_{32}J_2$ and we use this value to calculate a third increment

$$J_3 = kf(u^n + a_{31}J_1 + a_{32}J_2) \ .$$

Now with three increments we can estimate another value for $v(t)$ at the intermediate time $t = t_n + c_4 k$ such that $c_4 = a_{41} + a_{42} + a_{43}$, as $v(t) \approx u^n + a_{41}J_1 + a_{42}J_2 + a_{42}J_3$ and so on. Finally, the value u^{n+1} results form a linear combination of all the Js, i.e.

$$u^{n+1} = u^n + k\sum_{l=1}^{p} b_l J_l \ , \tag{4.53}$$

where the formulas for the Js can be formally written as

$$\begin{aligned} J_1 &= kf(u^n, t_n) \\ J_i &= kf\Big(u^n + k\sum_{l=1}^{i-1} a_{il}J_l, t_n + c_i k\Big) \qquad \text{for} \quad i = 2\cdots p \ , \end{aligned} \tag{4.54}$$

and we require $c_i = \sum_{l=1}^{i-1} a_{il}$. This is the generic formula for a p-stage RK method.

It is interesting to note that if one blindly starts with formulas (4.53)-(4.54), *consistency* of the scheme alone imposes the constraint $c_i = \sum_{l=1}^{i-1} a_{il}$ even if no formal relationship between the intermediate times is imposed.

The coefficients of the RK methods are calculated by imposing the order of the scheme, i.e. the terms in the *Taylor* expansion of Eq.(4.53) and (4.54) must be identical to their *Taylor series* counterparts from Eq. (4.46). The constraints imposed by identifying terms in both expansion does not completely determine the RK coefficients, and one have the freedom of imposing supplementary requirements. This is useful for example to minimise the factor in front of the truncation error.

Do not think it is and easy business: the calculation are unfortunately quite involved [12] we refer to Ralston & Rabinowitz (2001) for more details. Do not further assume that a p-stage RK method gives you necessarily a p order method. This is only true for $p \leq 3$ and for $p \geq 4$ we only have the general results that the number of stages is greater or equal to the order.

4.6 A first look at numerical instabilities

Let's consider now a simple ODE that will be one of our favourite test case for numerical integration methods

$$\dot{v}(t) = -\alpha\, v(t) \qquad \text{with} \tag{4.55}$$

$$v(0) = v^0 \ , \tag{4.56}$$

[12] Try to find all the order 3 RK methods and their truncation error and you will understand ...

where v^0 is the "initial" value and α is a positive constant.

The solution to Eq. (4.55) is $v(t) = v^0\, \mathrm{e}^{-\alpha\, t}$ and we expect from any good numerical integration method to approach this solution as close as we want. We already know for the *Euler* method that it is the case when $k \to 0$ but we might wonder what is going on when $k \neq 0$. Indeed, we have shown in the previous section that if $k \to 0$ the *Euler* method is convergent. We are facing now a different kind of problem. k is fixed as it should be in the real world and $t \to \infty$. In this situation we want k to be as large as possible for a prescribed precision because we want our program to go as fast as possible. A small k value means a large number of integration steps and thus an increasing computing time : we do not want to put too small a value of k. Can we use any k value without jumping into troubles ? Unfortunately, we cannot.

4.6.1 Explicit method

For the particular case of the *Euler*'s method we can actually obtain the numerical solution explicitly. In this particular case the *Euler* method is given by

$$\frac{u^{n+1} - u^n}{k} = -\alpha\, k\, u^n \ , \tag{4.57}$$

which is a recurence relation between the u^n that can be solved explicitly

$$u^n = (1 - \alpha\, k)^n\ u^0 \ , \tag{4.58}$$

where u^0 is the numerical approximation to v^0. Remember that we can have $u^0 \neq v^0$ (e.g. if $v^0 = \frac{1}{10}$ it cannot be represented exactly, see chapter 2).

We have $\left|\dfrac{u^{n+1}}{u^n}\right| = |1 - \alpha\, k|$ and consequently we have two different cases according to the value of $1 - \alpha\, k$

$$\left\{\begin{array}{llll} \text{if} & 1 - \alpha\, k > 0 & \text{then} & \left|\frac{u^{n+1}}{u^n}\right| = 1 - \alpha\, k \\ & & & \\ \text{if} & 1 - \alpha\, k < 0 & \text{then} & \left|\frac{u^{n+1}}{u^n}\right| = \alpha\, k - 1 \end{array}\right. . \tag{4.59}$$

We know that in any case the numerical solution must be bounded because the actual solution $v(t)$ is. Consequently, we must require that $\left|\frac{u^{n+1}}{u^n}\right| < 1$.

When $1 - \alpha\, k > 0$ this is ok because $\alpha\, k$ is positive and $1 - \alpha\, k$ is always lower than 1. So we do not have any problem whenever our step verifies $k < \frac{1}{\alpha}$.

When $1 - \alpha\, k < 0$, we must require $k < \frac{2}{\alpha}$ for our numerical solutions to be bounded. However, since we also have $\frac{u^{n+1}}{u^n} = 1 - \alpha\, k$ we see that even if this condition on k is fulfilled, $\frac{u^{n+1}}{u^n}$ is negative meaning that the values

u^n are actually oscillating ! As far as $\alpha\, k - 1$ is lower that 1 the u^n do not diverge to ∞ but still we are facing an ugly problem which definitely is not reflected by the actual behaviour of the solution $v(t)$.

Let's summarise our results. For the *Euler* explicit method we have

1. If $k < \frac{1}{\alpha}$, the u^n values are bounded,
2. if $\frac{1}{\alpha} < k < \frac{2}{\alpha}$, the u^n values are bounded *but they are oscillating* and the numerical results have no usefull purpose,
3. if $k > \frac{2}{\alpha}$ we are in the worse situation where the u^n values diverge to ∞ while they are oscillating !

It is time to introduce the notion of *absolute stability*. When k is fixed and $t \to \infty$ if the numerical solution of the test case expressed by Eq. (4.57) remains bounded the numerical method under consideration is then said to be *absolutely stable*. Note that for explicit *Euler* this is achieved when $k < \frac{2}{\alpha}$. For $\frac{1}{\alpha} < k < \frac{2}{\alpha}$, though the method is said to be absolutely stable, the numerical results are not useful! This is just a subtlety that one must keep in mind: our numerical results might remain bounded as they should *and* be crappy at the same time. Stability (of any kind) is *extremely* important for a numerical method. Unfortunately, even if it will be our main concern there could be other type of difficulties beyond stability.

4.6.2 Implicit method

The *Euler* implicit method can be written like this

$$\frac{u^{n+1} - u^n}{k} = -\alpha\, k\, u^{n+1} \ . \tag{4.60}$$

We can solve this equation right away to obtain

$$u^n = \left(\frac{1}{1 + \alpha\, k}\right)^n u^0 \ . \tag{4.61}$$

We remark that $\left|\frac{u^{n+1}}{u^n}\right| = \left|\frac{1}{1+\alpha\, k}\right|$ is always lower than 1 since $1 + \alpha\, k > 1$ for positive k. Consequently, the numerical values u^n for a fixed k will be *unconditionally* bounded. The *Euler* implicit method is unconditionally absolutely stable. Note also that since $1+\alpha\, k > 0$ we cannot have oscillating behaviour in our results.

This illustrates the advantage of the implicit methods with respect to the explicit ones. However there is a price to this unconditional stability. If the function f is not linear like in our test-case, and in many real situations it is not, we have to solve for a non-linear equation to obtain u^{n+1} out of u^n even for the simple *Euler* method.

As a final warning, keep in mind that stability means that truncation errors are not amplified without bounds, but they may be large *per se!* In other words, *accuracy* of the scheme, i.e. the magnitude of the discrepancy between the true solution and the numerical one, is governed by the truncation (or one-step) error, which in turns depends on the time step: even if the scheme is stable, using a too large time step may cause a too large error and a useless computation! So at the end remind that without absolute stability the computation may become evidently useless after a few time steps, but do not be too overconfident on it.

4.7 Multi-step methods: zero-stability and convergenge

For a generic one-step method we found in Eq. (4.36) a bound containing an amplification factor $\approx |1 + k\bar{K}|^{n-m}, 1 \leq m \leq n$ which turns out to be bounded as $k \to 0$, $n \to \infty$ since it goes like e^{KT}. Imagine instead to have an amplification factor like $|2 + k\bar{K}|^{n-m}$: this is clearly unbounded in the same limit. To be clear, because we are dealing with upper bounds, this *do not* yet prevents the global error to converge, but it can at least cast some doubts about it.

This observation leads as to analyze more carefully the convergence of more schemes more complex then one-step methods. Let's talk about *multi-step methods* remaining for simplicity in the linear case (4.1) $\dot{v}(t) = f\left(v(t), t\right)$. A *multi-step linear method* is a recipe that allows to get u^{n+1} using the previous $p+1$ values u^n, u^{n-1} $\ldots u^{n-p}$, which in particular can be written as

$$u^{n+1} + \sum_{r=0}^{p} \alpha_r u^{n-r} = k\lambda \left(\beta_{-1} u^{n+1} + \sum_{r=0}^{p} \beta_r u^{n-r} \right) \tag{4.62}$$

OR WHATEVER ... FOLLOWS AN "HANDWAWED" PROOF OF THE THEOREM CONSISTENCY + STABILITY -¿ CONVERGENCE

Chapter 5

Partial differential equations: the basics

5.1 Introduction

Solving partial differential equations (PDEs) is one of the most ubiquitous problems an (astro-) physicist (or mathematician, or whatever individual pretending doing science) must face. PDEs are a large universe, and in these notes we will present just the most basic examples and methods related to the most common problems. At lists for some classes of problems PDEs can be viewed as a collection of ODEs, but this is only partially true because switching from ODEs to PDEs introduces some qualitatively new difficulties. This is specially true for what concerns one of the most misunderstood, controversial and useful concepts, the one of stability. We will try to explain all this in the most practical and useful way.

5.1.1 Some mathematical properties

Basic classification: hyperbolic, parabolic, elliptic.

Initial-value problems / boundary value problems or both

Linear and nonlinear equations

TO BE DONE

5.2 An initial-value problem: the advection equation

Let $v(x,t)$ be a function of time and one space variable, with

$$\begin{aligned} & x \in [0, L] \\ & t \in [0, T], \qquad \text{(possibly } T \to \infty) \end{aligned}$$

which satisfies the *advection equation*:

$$\partial_t v + \alpha \partial_x v = 0 \quad , \qquad\qquad \alpha = \text{const} \tag{5.1}$$

We will consider the following *initial-value problem*: let assign an initial condition (initial field profile) $v(x,0) \equiv v^0(x)$ and determine the field $v(x,t)$ at any time t.

For this particularly simple equation we know the exact solution:

$$v(x,t) = v^0(x - \alpha t) \tag{5.2}$$

which means that the initial profile is simply displaced (*advected*) toward the right if $\alpha > 0$ or the left if $\alpha < 0$, with an *advection velocity* α, like in Fig. (5.1). In the following, if not otherwise specified, we will always take for simplicity $\alpha > 0$.

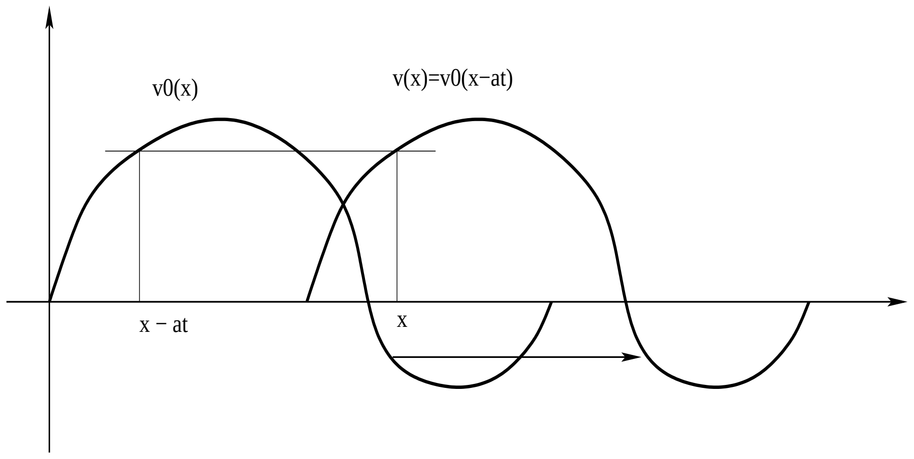

Figure 5.1: Evolution of an initial profile $v^0(x)$ obeying Eq. (5.1)

From the numerical point of view, the problem of looking for a solution of this equation is not so different from what we already examined for ODEs: look for an approximate solution u which can be determined with a deterministic algorithm in a finite number of steps. What changes here is just the number of variables and the fact that the initial condition is not a collection of numbers but a function, i.e. an infinite collection of numbers over which a supplementary (differential) operator is applied.

In the following of this chapter, we will use the advection equation as the main example to illustrate the basic concepts and pitfalls associated to the resolution of PDEs.

5.3 The most intuitive approach: finite differences

We can proceed on the same line as we did for ODEs, building a discretization over suitable intervals both in time and space, and approximate the differential operators by their finite differences counterparts (*finite differences* method). In fact it is quite common to stay with a finite-differences-like discretization for the time, while for what concerns the space there are many possible options, the most common being finite differences, finite volumes, finite elements, spectral methods (i.e. expansion over a basis of functions) .... The simplest (or more intuitive) method is nevertheless finite differences, while in "real" problems the choice of the best treatment for the spatial variable(s) is often a compromise between different needs: dealing with simple or awkward boundary conditions (i.e. conditions the function v must satisfy on the boundaries of the spatial domain), precision, rapidity, discretization over complicated geometries, and last but not least simplicity of coding.

Let us proceed then with finite differences. Build a discretization (*grid*) of the space and time intervals:

$$x_j = j\Delta x \quad j = 0, 1, \dots M \quad \Delta x = L/M \equiv h \tag{5.3}$$

$$t_n = n\Delta t \quad n = 0, 1, \dots N \tag{5.4}$$

$\Delta x \equiv h$ is the *mesh size*, $\Delta t \equiv k$ is the *time step*. Discrete times and positions will be then indicated as x_j, t_n or simply j, n. We consider then a finite collection of numbers: $\{u_j^n\}$, $j = 0 \dots M$, $n = 0 \dots T/k$ which approximate the values of the exact solution v on the discrete grid: $v(x_j, t_n) \equiv v_j^n \approx u_j^n$. We remind here the usual notation we use, by indicating with v the exact solution and u the approximate one. Moreover, we place time indices as superscripts, and space indices as subscript (this was the reason why in ODEs we used the weird superscript notation for the time).

We can then replace the derivatives with their finite differences approximations, which will be exact in the limits $h \to 0$, $k \to 0$. For example when using backward differences for both time and space we get

$$(\partial_t v)_j^n \approx \frac{u_j^{n+1} - u_j^n}{k} \tag{5.5}$$

for the time derivative, and

$$(\partial_x v)_j^n \approx \frac{u_j^n - u_{j-1}^n}{h} \tag{5.6}$$

for the space derivative. Equation (5.1) will be then replaced by the approximation on the grid:

$$\frac{u_j^{n+1} - u_j^n}{k} + \alpha \frac{u_j^n - u_{j-1}^n}{h} = 0. \tag{5.7}$$

which can be rewritten as

$$u_j^{n+1} = u_j^n \left(1 - \alpha \frac{k}{h}\right) u_j^n + \alpha \frac{k}{h} u_{j-1}^n. \tag{5.8}$$

This is our *numerical scheme*, i.e. a closed formula that with a finite number of operations allows the calculation of all $\{u_j\}$ at the instant t_{n+1} once the corresponding set is known at the instant t_n. It is then suitable for being programmed in a computer to solve the initial-value problem .

For what concerns the time advancing, we use the same terminology as for ODEs: this scheme is then called *explicit Euler* in time. Moreover, using the one-sided finite differences approximation (5.6) for the space derivative qualifies the scheme as *upwind* (if $\alpha > 0$) in space. The term *upwind* (which is restricted to equations that describe the displacement of a profile), means that the approximate solution u_j^{n+1} at position j is built using the information at positions $j, j-1$, i.e. on the side from which the signal is coming (since the solution profile propagates from left to right for $\alpha > 0$). This way of doing is far from being innocent and has a deep influence on the stability of the scheme as it will be seen in the following.

In general an explicit Euler scheme for Eq. (5.1) reads

$$\frac{u_j^{n+1} - u_j^n}{k} = \Phi(\{u^n\})_j \tag{5.9}$$

where Φ is a generic function on the collection of finite values $\{u_j^n\}$, $j = 0 \dots M$, taken all at the same time t_n that approximates the differential operator $-\alpha \partial_x v$. We can then use any possible way of approximating the spatial derivative using the function values on the grid points, and this will be a perfectly legitimate as far as the derivative is reproduced when the mesh size is reduced. For example, other possible approximations for the space derivatives can be:

$$(\partial_x v)_j^n \approx \frac{u_{j+1}^n - u_j^n}{h} \tag{5.10}$$

(one-sided *downwind* finite differences, which is just the approximation that searches on the side to which the signal is going) or

$$(\partial_x v)_j^n \approx \frac{u_{j+1}^n - u_{j-1}^n}{2h} \tag{5.11}$$

(centered finite differences)

and obtain this way the two following explicit Euler schemes:

$$u_j^{n+1} = \left(1 + \alpha \frac{k}{h}\right) u_j^n - \alpha \frac{k}{h} u_{j+1}^n \tag{5.12}$$

(Euler downwind)

or

$$u_j^{n+1} = u_j^n - \alpha \frac{k}{2h} (u_{j+1}^n - u_{j-1}^n). \tag{5.13}$$

(Euler centered)

If we do more complicated things on the spatial differential operator (i.e. expanding the spatial profile over a function basis and then using the coefficients of the expansion instead of the grid values) this will not be anymore a finite differences schemes in space, but it will still be Euler explicit in time.

To be more general, we can (at least in principle) combine *any* time discretization with any space discretization, for example we can do an *implicit Euler* in time by writing generically:

$$\frac{u_j^{n+1} - u_j^n}{k} = \Phi(\{u^{n+1}\})_j \tag{5.14}$$

where the approximation for the differential operator on the rhs is is taken at time t_{n+1} instead of at t_n, or building a trapezoidal *Cranck-Nicolson* scheme as

$$\frac{u_j^{n+1} - u_j^n}{k} = \frac{1}{2}\left(\Phi(\{u^n\})_j + \Phi(\{u^{n+1}\})_j\right). \tag{5.15}$$

which combines approximations for the differential operator at both times t_n, t_{n+1}.

Just to give a less trivial example one can take the *leap-frog* approximation for the time derivative and combine it with a centered spatial approximation:

$$\frac{u_j^{n+1} - u_j^{n-1}}{2k} + \alpha\frac{u_{j+1}^n - u_{j-1}^n}{2h} = 0. \tag{5.16}$$

(Leap-frog centered)

All these possibilities are *not* equivalent and lead to different numerical results (which can be really bad!!!), some being stable, some unstable, some very precise and some less. In the following we will see why some possibilities are admissible and some are not.

5.3.1 PDEs like vector ODEs

Take the explicit Euler upwind scheme written as in (5.8).

$$u_j^{n+1} = (1-p)u_j^n + pu_{j-1}^n \tag{5.17}$$

where here and in the following we introduced the fundamental parameter for the numerical discretization of the advection equation:

$$p \equiv \alpha\frac{k}{h}. \tag{5.18}$$

Putting aside for the moment what happens at the boundaries $j = 0$,

$j = M$ we can write in in matrix form :

$$\begin{pmatrix} \dots \\ u_{j-1}^{n+1} \\ u_j^{n+1} \\ u_{j+1}^{n+1} \\ \dots \end{pmatrix} = \begin{pmatrix} & & & \dots & & & \\ \dots & p & (1-p) & 0 & 0 & 0 & \dots \\ \dots & 0 & p & (1-p) & 0 & 0 & \dots \\ \dots & 0 & 0 & p & (1-p) & 0 & \dots \\ & & & \dots & & & \end{pmatrix} \begin{pmatrix} \dots \\ u_{j-1}^{n} \\ u_j^{n} \\ u_{j+1}^{n} \\ \dots \end{pmatrix} \tag{5.19}$$

which means that the scheme can be actually be written as a *vector ODE* by grouping the collections of the values $\{u_j^n\}$, $j = 0 \dots M$ into a vector $\mathbf{u}^n$:

$$\mathbf{u}^{n+1} = A\mathbf{u}^n \tag{5.20}$$

where the matrix A $(M+1) \times (M+1)$ is the one written in (5.19)

In fact *any* finite differences scheme for PDEs can be actually written like a vector ODE in time where the finite number of function values in space constitute the vector over which the ODE operates. Given a generic one-step time-advancing scheme and a generic spatial discretization for which we have the general formula

$$u_j^{n+1} = u_j^n + k\Psi\left(\{u^n\}, \{u^{n+1}\}, t_n, k, h\right)_j \tag{5.21}$$

(Ψ being some operator whose form depends on the approach used to discretize the spatial differential ones, and the scheme being explicit if the dependence on the time t_{n+1} on the rhs is dropped), it can be cast in vector form :

$$\mathbf{u}^{n+1} = \mathbf{u}^n + k\boldsymbol{\Psi}(\mathbf{u}^n, \mathbf{u}^{n+1}, t_n, k, h) \tag{5.22}$$

where $\boldsymbol{\Psi}$ denotes the same operator by acting on the vector in full. From a formal point of view, the introduction of multistep methods does not add any supplementary problem.

5.3.2 Examples of implicit schemes

We saw that for ODEs "implicit scheme" means "in a form that is not already explicitly solved", and for a scalar linear equation this is pretty trivial. For PDEs the idea is the same, but in the best case we are led to the resolution of a linear system. Let's take for example the explicit Euler upwind scheme (5.8). We can write its implicit version by taking the same approximation for the space derivative but with function values at time t_{n+1} instead of at t_n, i.e.:

$$(\partial_x v)_j^n \approx \frac{u_j^{n+1} - u_{j-1}^{n+1}}{h} \tag{5.23}$$

and build the following scheme:

$$\frac{u_j^{n+1} - u_j^n}{k} + \alpha\frac{u_j^{n+1} - u_{j-1}^{n+1}}{h} = 0. \tag{5.24}$$

i.e.

$$(1+p)u_j^{n+1} - pu_{j-1}^{n+1} = u_j^n \tag{5.25}$$

which can be cast as a vector equation :

$$B\mathbf{u}^{n+1} = \mathbf{u}^n \tag{5.26}$$

where the matrix A is given by

$$B = \begin{pmatrix} & & & \dots & & & \\ \dots & -p & (1+p) & 0 & 0 & 0 & \dots \\ \dots & 0 & -p & (1+p) & 0 & 0 & \dots \\ \dots & 0 & 0 & -p & (1+p) & 0 & \dots \\ & & & \dots & & & \end{pmatrix}. \tag{5.27}$$

To get $\mathbf{u}^{n+1}$ from $\mathbf{u}^n$ one is then led to solve a linear system. In general a (linear) one-step implicit scheme for a linear equation will read as

$$B\mathbf{u}^{n+1} = A\mathbf{u}^n + \mathbf{c}(t_n, k) \tag{5.28}$$

with two matrices A and B and a possible time-dependent source $\mathbf{c}$ (which can depend on some intermediate time between t_n, t_{n+1}). The source structure may be evident from the equation itself or contain information from the *boundary conditions* as will be seen in the following Section. As a further example, the *Cranck-Nicolson* scheme for Eq. (5.1) with a centered approximation for the spatial derivative, i.e. :

$$\frac{u_j^{n+1} - u_j^n}{k} + \frac{\alpha}{2}\left(\frac{u_{j+1}^{n+1} - u_{j-1}^{n+1}}{2h} + \frac{u_{j+1}^n - u_{j-1}^n}{2h}\right) = 0 \tag{5.29}$$

in vector form reads as (5.28) with A and B given by

$$A = \begin{pmatrix} & & & \dots & & & \\ \dots & p/4 & 1 & -p/4 & 0 & 0 & \dots \\ \dots & 0 & p/4 & 1 & -p/4 & 0 & \dots \\ \dots & 0 & 0 & p/4 & 1 & -p/4 & \dots \\ & & & \dots & & & \end{pmatrix} \tag{5.30}$$

$$B = \begin{pmatrix} & & & \dots & & & \\ \dots & -p/4 & 1 & p/4 & 0 & 0 & \dots \\ \dots & 0 & -p/4 & 1 & p/4 & 0 & \dots \\ \dots & 0 & 0 & -p/4 & 1 & p/4 & \dots \\ & & & \dots & & & \end{pmatrix} \tag{5.31}$$

and a vector $\mathbf{c}$ uniquely dependent on the boundary conditions as we will see in the following.

If the equation is *not* linear (as all "real" equations of mathematical physics are), the problem is much more complex because instead of solving a linear system we end up with the resolution of a vector algebraic nonlinear equations, which involves the use of some iterative methods (i.e. Newton-Raphson or something similar), in which the imposed tolerance on the converge level introduces some supplementary error. So why bother with implicit schemes? Because like in ODEs explicit schemes are in some cases unstable or need unacceptable small time steps, or do not have the required conservation properties (i.e. momentum or energy) that we may *want* to be preserved by the numerics. But because implicit schemes need always some more computational work, and even when stable you cannot usually increase the timestep without bounds (because the scheme, even if stable, will be too inaccurate due to a too large truncation error), the need for an implicit scheme must be carefully evaluated. So depending on the case it can be better an explicit scheme which needs a small time step instead of an unconditionally stable implicit one.

5.4 Boundary conditions

To be considered a closed algorithm, a formula of the kind (5.8) which couples different space indices j, $j-1$ (or more generally (5.21) including its vector counterpart (5.22)), must be supplemented by some treatment close to the "boundaries" where the general formula contains meaningless indices, $j < 0$ or $j > M$. While a generic initial-value ODE needs just an initial condition, for solving a PDE we need such a condition (in the form $u_j^0 = v^0(x_j)\ \forall\, j$) but we must also know how the profile "ends" at the geometric boundaries where the differential operators are well defined only on one side. In the discrete representation this translates to the impossibility of blindly using the finite difference approximations for the space derivatives.

Writing from example Eq. (5.8) explicitly:

$$\begin{aligned}
u_0^{n+1} &= (1-p)u_0^n + pu_{-1}^n \\
u_1^{n+1} &= (1-p)u_1^n + pu_0^n \\
&\dots \\
u_{j-1}^{n+1} &= (1-p)u_{j-1}^n + pu_{j-2}^n \\
u_j^{n+1} &= (1-p)u_j^n + pu_{j-1}^n \\
u_{j+1}^{n+1} &= (1-p)u_{j+1}^n + pu_j^n \\
&\dots \\
u_{M-1}^{n+1} &= (1-p)u_{M-1}^n + pu_{M-2}^n \\
u_M^{n+1} &= (1-p)u_M^n + pu_{M-1}^n
\end{aligned} \tag{5.32}$$

it appears that the equation in the first line has no meaning because it contains the value u^n_{-1}, which does not exist because there is no point x_{-1}. One has to introduce then some "closures", in the form of *boundary conditions* (b.c). These are not specific to the numerical discretization, since in general any PDE (5.1) needs some b.c. in order to get a unique solution. Take for example the displacement of a vibrating string guitar from the rest position, which is also described by some PDE. The vibration will be clearly different if we prescribed fixed both ends or if one of the ends is free or constrained to some special trajectory. So first of all we need to ask: what are my *physical* b.c.? Fixed ends, moving ends in some specified way, or something else? In the case of our advection equation, let's remind that the exact solution is a profile propagating from left to right if $\alpha > 0$. So it seems a little bit strange to prescribe the boundaries, because they should evolve "automatically" with the natural evolution of the profile[1]. Something "innocent" we can do is prescribing for example *periodic* b.c., i.e. $v(x, t_n) = v(x + L, t_n)$. In particular this implies in the numerics that $u^n_0 = u^n_M \forall\, n$, so the value u^n_M does need to be calculated and the number of coupled equations reduce from $M + 1$ to M. Moreover, "meaningless" values like u^n_{-1} are replaced by periodicity by some others, i.e. $u^n_{-1} = u^n_{M-1}$.

Once this is done, the system (5.32) in vector form reads :

$$\begin{pmatrix} u_0^{n+1} \\ u_1^{n+1} \\ \dots \\ u_{j-1}^{n+1} \\ u_j^{n+1} \\ u_{j+1}^{n+1} \\ \dots \\ u_{M-2}^{n+1} \\ u_{M-1}^{n+1} \end{pmatrix} = \begin{pmatrix} (1-p) & 0 & \dots & 0 & 0 & 0 & \dots & 0 & p \\ p & (1-p) & \dots & 0 & 0 & 0 & \dots & 0 & 0 \\ & & & & \dots & & & & \\ 0 & 0 & \dots & (1-p) & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & \dots & p & (1-p) & 0 & \dots & 0 & 0 \\ 0 & 0 & \dots & 0 & p & (1-p) & \dots & 0 & 0 \\ & & & & \dots & & & & \\ 0 & 0 & \dots & 0 & 0 & 0 & \dots & (1-p) & 0 \\ 0 & 0 & \dots & 0 & 0 & 0 & \dots & p & (1-p) \end{pmatrix} \begin{pmatrix} u_0^n \\ u_1^n \\ \dots \\ u_{j-1}^n \\ u_j^n \\ u_{j+1}^n \\ \dots \\ u_{M-2}^n \\ u_{M-1}^n \end{pmatrix} \tag{5.33}$$

the relevant matrix being non-zero in the main diagonal, in the strip below and in the right-upper corner (do not miss that!!!).

The reader can check that when employing the centered approximation (5.13) for the space derivative, the corresponding matrix is *tridiagonal* with

[1]in fact the case of advection equation is somehow special for what concerns b.c.: for other equations like the diffusion or the wave equation it is more natural to impose b.c without resorting to infer some information from the exact solution that you are *not* supposed to know when numerically integrate

corners :

$$\begin{pmatrix} 1 & -p/2 & 0 & \dots & 0 & 0 & 0 & \dots & 0 & p/2 \\ p/2 & 1 & -p/2 & \dots & 0 & 0 & 0 & \dots & 0 & 0 \\ & & & & & \dots & & & & \\ 0 & 0 & 0 & \dots & 1 & -p/2 & 0 & \dots & 0 & 0 \\ 0 & 0 & 0 & \dots & p/2 & 1 & -p/2 & \dots & 0 & 0 \\ 0 & 0 & 0 & \dots & 0 & p/2 & 1 & \dots & 0 & 0 \\ & & & & & \dots & & & & \\ 0 & 0 & 0 & \dots & 0 & 0 & 0 & \dots & 1 & -p/2 \\ -p/2 & 0 & 0 & \dots & 0 & 0 & 0 & \dots & p/2 & 1 \end{pmatrix} \tag{5.34}$$

(we will nevertheless discover that the explicit Euler scheme is unstable with such a centered approximation, and in practice we need an implicit scheme).

Even not meaningful, we can nevertheless look at our schemes in case we *want* to impose *fixed boundary conditions*. In general, this implies that the values of v (and henceforth u) at both ends remain fixed to specified values, i.e.:

$$\begin{aligned} u_0^n &= \gamma_a \quad \forall\, n \\ u_M^n &= \gamma_b \quad \forall\, n \end{aligned} \tag{5.35}$$

which means that the first and the last equations of (5.32) must *not* be integrated, because their result is imposed from the exterior. Moreover, in the second equation of (5.32) the value u_0^n must be replaced by γ_a at any time. The number of simultaneous equations to be solved reduces then from $M+1$ to $M-1$, and with an upwind space discretization we write in matrix form :

$$\begin{pmatrix} u_0^{n+1} \\ u_1^{n+1} \\ \dots \\ u_{j-1}^{n+1} \\ u_j^{n+1} \\ u_{j+1}^{n+1} \\ \dots \\ u_{M-2}^{n+1} \\ u_{M-1}^{n+1} \end{pmatrix} = \begin{pmatrix} (1-p) & 0 & \dots & 0 & 0 & 0 & \dots & 0 & 0 \\ p & (1-p) & \dots & 0 & 0 & 0 & \dots & 0 & 0 \\ & & & & \dots & & & & \\ 0 & 0 & \dots & (1-p) & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & \dots & p & (1-p) & 0 & \dots & 0 & 0 \\ 0 & 0 & \dots & 0 & p & (1-p) & \dots & 0 & 0 \\ & & & & \dots & & & & \\ 0 & 0 & \dots & 0 & 0 & 0 & \dots & (1-p) & 0 \\ 0 & 0 & \dots & 0 & 0 & 0 & \dots & p & (1-p) \end{pmatrix} \begin{pmatrix} u_0^n \\ u_1^n \\ \dots \\ u_{j-1}^n \\ u_j^n \\ u_{j+1}^n \\ \dots \\ u_{M-2}^n \\ u_{M-1}^n \end{pmatrix} + \begin{pmatrix} p\gamma_a \\ 0 \\ \dots \\ 0 \\ 0 \\ 0 \\ \dots \\ 0 \\ 0 \end{pmatrix} \tag{5.36}$$

i.e. the non-zero value in the upper right corner is now 0 and the left b.c. γ_a appears like a source in the rhs. Funny enough, the right b.c. does not appear at all and can be dropped. From the numerical point of view (but not really from the physical one) this can be understood by noticing that this upwind scheme needs values coming form the left in approximating the derivative, so it is reasonable that the rightmost value is never needed. The resulting matrix is diagonal with a lower band. Now when employing the centered approximation (5.13) for the space derivative the scheme reads :

$$\begin{pmatrix} u_0^{n+1} \\ u_1^{n+1} \\ u_2^{n+1} \\ \dots \\ u_{j-1}^{n+1} \\ u_j^{n+1} \\ u_{j+1}^{n+1} \\ \dots \\ u_{M-2}^{n+1} \\ u_{M-1}^{n+1} \end{pmatrix} = \begin{pmatrix} 1 & -p/2 & 0 & \dots & 0 & 0 & 0 & \dots & 0 & 0 \\ p/2 & 1 & -p/2 & \dots & 0 & 0 & 0 & \dots & 0 & 0 \\ 0 & p/2 & 1 & \dots & 0 & 0 & 0 & \dots & 0 & 0 \\ & & & & & \dots & & & & \\ 0 & 0 & 0 & \dots & 1 & -p/2 & 0 & \dots & 0 & 0 \\ 0 & 0 & 0 & \dots & p/2 & 1 & -p/2 & \dots & 0 & 0 \\ 0 & 0 & 0 & \dots & 0 & p/2 & 1 & \dots & 0 & 0 \\ & & & & & \dots & & & & \\ 0 & 0 & 0 & \dots & 0 & 0 & 0 & \dots & 1 & -p/2 \\ 0 & 0 & 0 & \dots & 0 & 0 & 0 & \dots & p/2 & 1 \end{pmatrix} \begin{pmatrix} u_0^n \\ u_1^n \\ u_2^n \\ \dots \\ u_{j-1}^n \\ u_j^n \\ u_{j+1}^n \\ \dots \\ u_{M-2}^{n+1} \\ u_{M-1}^n \end{pmatrix} + \begin{pmatrix} \frac{p}{2}\gamma_a \\ 0 \\ 0 \\ \dots \\ 0 \\ 0 \\ 0 \\ \dots \\ 0 \\ -\frac{p}{2}\gamma_b \end{pmatrix} \tag{5.37}$$

where the matrix is purely tridiagonal and now *both* boundary conditions enter (as a source) into the numerical scheme.

The reader at this point must realize that something is going wrong. How is possible that according to the numerical scheme we adopt, we use one or both (or possibly none) values at the border? In fact this is not possible, which means that some schemes are good for dealing with borders and some other are not. Which is good depends in turn on the properties of the equation itself. In fact the advection equation is a member of a general class called *hyperbolic systems*. Loosely speaking, they are characterized by a finite propagation speed (α in our case) in a given direction, which are the propagation speed and direction of the information as well. As a consequence, the solution $v(x,t)$ at a given point and a given time depends on the solution values $v(x',t')$ at all previous times $t' < t$ but not at all points in space, only at points $x - \alpha t' < x' < x$ for a given t'. This implies that for $\alpha > 0$, i.e. for a profile propagating to the right, the solution at a given point is influenced by the previous solutions, but only at some points on the left of the actual one. For this reason it is perfecly legal to set the leftmost boundary but not the rightmost, and all schemes which need to fix a rightmost boundary are doomed to fail (unless you are doing periodic b.c. but this is another story).

ALL THE PAGE IS BEING REPHRASED

5.5 Truncation errors and consistency

So far we approximated our equation by taking some finite differences on a discrete space-time grid instead of differential operators. Sounds good... but is there a way to know whether we are doing well, or even how much error we introduce with this procedure? Well, since a PDE can be written as a vector ODE, we can proceed as we did for ODEs. As a first measure of the goodness of our approximation is the *one-step error*, given by the difference between the evolution of the field driven by the true equation $v(x_n, t_n) \equiv v_j^n$

and the one calculated with the numerical scheme. This difference will be taken on one time step, i.e. on a time increment of k, taking care to assign to the numerical solution from which we start the values of the continuous exact solution on the spatial grid. In short, we impose $u_j^n = v_j^n$ and do one step with both the exact equation and the numerical scheme.

Let us consider first the explicit Euler upwind (5.8):

$$u_j^{n+1} = (1-p)u_j^n + pu_{j-1}^n. \tag{5.38}$$

We write then the *one-step error* (from the step n to the step $n+1$ at point x_j) as:

$$e_j^{n+1} = u_j^{n+1} - v_j^{n+1} \tag{5.39}$$

under the condition $u_j^n = v_j^n$.

Actually the error must be defined in some norm, e.g. L^2 or L^∞ (sup norm) or you can even consider a collection of one-point problems and take the norm $|e_j^n|$ at each point. We now proceed to evaluate now e_j^n without taking any norm, because the resulting expression can give supplementary information beyond its magnitude (in particular the sign of e will be useful later).

$$\begin{aligned}
e_j^{n+1} &= u_j^{n+1} - v_j^{n+1} \\
&= \left[(1-p)u_j^n + pu_{j-1}^n\right] - v(x_j, t_n + \Delta t) \\
&= \left[(1-p)v_j^n + p\left(v_j^n - h{v'}_j^n + \frac{1}{2}h^2{v''}_j^n + \mathcal{O}(h^3)\right)\right] \\
&\quad - \left(v_j^n + k\dot{v}_j^n + \frac{1}{2}k^2\ddot{v}_j^n + \mathcal{O}(k^3)\right) \\
&= -k(\dot{v}_j^n + \alpha{v'}_j^n) - \frac{1}{2}k^2(\ddot{v}_j^n - \alpha\frac{h}{k}{v''}_j^n) + k(\mathcal{O}(h^2) + \mathcal{O}(k^2)) \\
&= -\frac{1}{2}\alpha k(\alpha k - h){v''}_j^n + k(\mathcal{O}(h^2) + \mathcal{O}(k^2))
\end{aligned} \tag{5.40}$$

where we Taylor-expanded the function v (dots indicate time derivatives and primes indicate space derivatives) and we used $u_j^n = v_j^n$: only *after* this substitution we are allowed to Taylor-expand in space and time the function u, which is defined over a discrete grid and the notion of derivative has no meaning for it. Moreover, we used the fact that v satisfies exactly the equation, so $\dot{v}_j^n + \alpha{v'}_j^n = 0$ and differentiating the initial equation we have $\ddot{v}_j^n = \alpha^2{v''}_j^n$.

We stress that we never made use of the exact solution : the calculation of the one-step error is in fact not based at all on the knowledge of the solution, but only on the form of the equation.

If we used instead the explicit Euler *downwind* (5.12), the error have been taken the form :

$$e^{n+1} = -\frac{1}{2}\alpha k(\alpha k + h){v''}_j^n + k(\mathcal{O}(h^2) + \mathcal{O}(k^2)). \tag{5.41}$$

In both cases the one-step error is $k\,(O(k)+O(h))$, the first term coming from the time discretization and the second from the spatial one. Following what we said for the ODEs, the scheme is then first order in time and space, the extra power in k being absorbed because to reach a finite time T we perform T/k time steps. This is typical of all initial-value problems evolving from an initial condition: in this case a numerical scheme will be $p-$ th order in time and $q-$ th order in space if the one-step error is $k\,(O(k^p)+O(h^q))$ at leading order. Dividing by 2 the time step or the mesh size has the consequence that the one-step precision is increased by factors respectively 2^p and 2^q, being evident the interest of employing high-order schemes. One must nevertheless carefully counterbalance the fact that higher-order schemes need generally more computational work per step. Moreover, in some conditions high-order schemes may actually introduce some undesirable features in the solution, i.e. spurious oscillations[2].

The form (5.42) is quite interesting, because it suggests that at leading order the error is *zero* if $\alpha k = h$ exactly. So for PDEs in some cases we can minimize the error not only the hard way, i.e. taking k, h as small as possible, but being smarter and making a good combined choice for the time step and the mesh size. This of course does not mean that the error is exactly zero, but the leading contributions will come at the next order and the method will be effectively of higher order (this procedure is not trivial for complicated schemes though...).

Coming to the local truncation error τ, just for the sake of clarity we write it down for both upwind (5.8) and downwind (5.12) Euler methods. This is straightforward if we use the scheme in the form:

$$
\begin{aligned}
\frac{u_j^{n+1}-u_j^n}{k} + \alpha\frac{u_j^n-u_{j-1}^n}{k} &= 0 \qquad \text{(upwind)}\\
\frac{u_j^{n+1}-u_j^n}{k} + \alpha\frac{u_{j+1}^n-u_j^n}{k} &= 0 \qquad \text{(downwind)}
\end{aligned}
$$

from which the truncation errors come out immediately by plugging in the exact solution:

$$
\begin{aligned}
\tau^{n+1} &= \frac{v_j^{n+1}-v_j^n}{k} + \alpha\frac{v_j^n-v_{j-1}^n}{k} \qquad \text{(upwind)}\\
\tau^{n+1} &= \frac{v_j^{n+1}-v_j^n}{k} + \alpha\frac{v_{j+1}^n-v_j^n}{k} \qquad \text{(downwind)}
\end{aligned}
$$

The use of the vector form results very convenient to highlight the formal similarities between PDEs and ODEs. Consider a generic PDE of the form

[2]exactly in the same way as using polynomials of too high order for doing Lagrange interpolation of a function: standard finite-differences schemes are in fact a sort of polynomial interpolation of a function which is known only on a discrete set of points.

$\partial_t v = F(v,t) = 0$ where F is a generic operator containing spatial derivatives, and let's suppose to have a generic scheme in vector form written as for vector ODEs : (5.22) :

$$\mathbf{u}^{n+1} = \mathbf{u}^n + k\mathbf{\Psi}(\mathbf{u}^n, \mathbf{u}^{n+1}, t_n, k, h)$$

the one-step error is $\mathbf{e}^{n+1} = \mathbf{u}^{n+1} - \mathbf{v}^{n+1}$ under the condition $\mathbf{u}^n = \mathbf{v}^n$, while the truncation error is

$$\boldsymbol{\tau}^{n+1} = \frac{\mathbf{v}(t_{n+1}) - \mathbf{v}(t_n)}{k} - \mathbf{\Psi}\left(\mathbf{v}^n, \mathbf{v}^{n+1}, t_n, k, h\right).$$

For the consistency requirement we now take two limits: we require $\boldsymbol{\tau} \to 0$ as $k \to 0$, $h \to 0$, and in this limit the usual relationship $\boldsymbol{e} \sim -k\boldsymbol{\tau}$ holds. Moreover, consistency implies $\mathbf{\Psi}\left(\mathbf{v}^n, \mathbf{v}^n, t_n, 0, =\right) = F(v^n, t_n)$ formally as for ODEs. We will be free of using the vector form or the pointwise form according to which is most convenient for our purposes.

As a matter of example, all explicit Euler schemes we presented so far (upwind, downwind, centered) are consistent since for the approximation of the time derivative :

$$\frac{v_j^{n+1} - v_j^n}{k} = \frac{v_j^n + k\dot{v}_j^n + \ldots - v_j^n}{k} = \dot{v}_j^n + \mathcal{O}(k) \to \dot{v}_j^n \quad \text{for } k \to 0$$

while for the spatial derivatives :

$$\frac{v_j^n - v_{j-1}^n}{h} = \frac{v_j^n - v_j^n + hv'^n_j + \ldots}{h} = v'^n_j + \mathcal{O}(h) \to v'^n_j \quad \text{for } h \to 0$$

$$\frac{v_{j+1}^n - v_j^n}{h} = \frac{v_j^n + hv'^n_j + \ldots - v_j^n}{h} = v'^n_j + \mathcal{O}(h) \to v'^n_j \quad \text{for } h \to 0$$

$$\frac{v_{j+1}^n - v_{j-1}^n}{2h} = \frac{v_j^n + hv'^n_j + \ldots - v_j^n + hv'^n_j + \ldots}{2h} = v'^n_j + \mathcal{O}(h^2) \to v'^n_j \quad \text{for } h \to 0$$

and in all these cases the local truncation error will reduce to :

$$\tau_j^{n+1} = \dot{v}_j^n + \alpha v'^n_j + (\mathcal{O}(k^p) + \mathcal{O}(h^q)) \to 0 \quad \text{for } k \to 0,\ h \to 0$$

5.6 Convergence and stability

At first it could seem that the issue of convergence and stability for PDEs follows the same lines as for ODEs, leading to the same conclusions. This is partially so and partially *not so* : the fact of having now two independent variables k, h that can go to zero independently makes the matter a bit more complicated, with the result that *for PDEs zero-stability usually cannot be established*, even for one-step methods. But before proceeding explaining this, we look at the notion of stability in a somehow rudimentary and intuitive way by exploring what the one-step and truncation errors can tell us besides consistency and order of the scheme.

5.6.1 One-step error and stability : loose ties

As for ODEs, a numerical scheme for PDEs will be useful if stable. Loosely speaking, stability means that the error on the solution will not be amplified without control. The one-step error can be used as a first measure to understand when this could happen or could not happen, and how not only the time but the spatial discretization (and their coupling) bears an influence on that. Consider as usual the advection equation $\partial_t v + \alpha \partial_x v = 0$ integrated with the explicit Euler upwind scheme (5.8) (remember that $p = \alpha k/h$)

$$u_j^{n+1} = (1-p)\, u_j^n + p u_{j-1}^n$$

for which the one-step error at the dominant order is

$$e_j^{n+1} = u_j^{n+1} - v_j^{n+1} \sim -\frac{1}{2}\alpha k(\alpha k - h) v''^n_j \tag{5.42}$$

Let us take for example a *sin* profile that according to the equation propagates to the right ($\alpha > 0$, and use e.g. periodic b.c.). If we choose the space and time steps in such a way that $\alpha k > h$ (i.e. $p > 0$), the error e will be positive where v'' is negative, i.e. preferentially at the maxima of the profile, while it will be negative at the minima. Now since by construction $u_j^n = v_j^n$, $e > 0$ implies $u_j^{n+1} > v_j^{n+1}$, i.e. at t_{n+1} the numerical solution is be larger than the exact one while at the previous time they were equal. In the present case it happens at the profile maxima, while at the profile minima since $e < 0$ the numerical solution will be smaller that the exact one. This means that *for $p > 0$ at each time step the numerical solution profile has the tendency to "spread" with respect to the exact one.* There are no upper bounds to this spreading, and the numerical solution can grow to infinity as time elapses : this is the typical *unstable behavior.* Incidentally, we say that at the leading order the truncation error of the procedure is *anti-dissipative.* If on the contrary $\alpha k < h$ (i.e. $p < 0$) the sign of e is just opposite, and the numerical profile will tend to "shrink" with respect to the exact one. This is in fact a *stable* behavior, because the deviation form the exact one is bounded, as the numerical solution will eventually collapse to zero[3]. In this case at leading order the truncation error is *dissipative.* The condition for "stability" $\alpha k < h$ means that in one time step, the profile translates of a quantity αk (remember that α is the advection velocity of the solution) smaller than a mesh size h. This is known as *CFL: Courant-Friedrichs-Lewy* condition, and essentially means that in one time step the profile must not

[3]this process could be considered as well a sort of instability, because in one's mind any systematic deviation can reasonably be thought as an unstable behavior. But semantically instability is something that leads to an *unbounded deviation*, and moreover numerically it is quite hard to obtain just a numerical solution that smoothly oscillates around the exact one. Least but not last, instability in a machine usually lead to infinities very short, while too much dissipation can be much more easily controlled.

be advected by more than a mesh size. This way, since the approximation on the spatial derivatives needs some information from the moving profile, this approximation is not "mixed up" from one time step to the other due to an insufficient resolution. This condition is relevant for upwind schemes, in which - loosely speaking - at a point the numerics is able to get the information from the solution coming towards this point.

One can consider that even if "stable", a numerical solution that while advecting the profile eventually dissipates it to zero is not so useful. This is the problem of stable but dissipative schemes. We will learn that most of the time a stable scheme is dissipative as well : this is acceptable if we are able to control the dissipation level and stop the computation before dissipating too much, which is part of the "numerical art".

NUMERICAL EXAMPLES ... SOME GRAPHS.

Let us take now the explicit Euler downwind scheme (5.12)

$$u_j^{n+1} = (1+p)\,u_j^n - p u_{j+1}^n$$

for which the one-step error at the dominant order is

$$e_j^{n+1} = u_j^{n+1} - v_j^{n+1} = -\frac{1}{2}\alpha k(\alpha k + h) v''^n_j \tag{5.43}$$

In this case the coefficient in front of the second derivative is always negative, and following the former discussion for a *sin* profile we are always in the conditions of "instability". This method is in fact unstable : from the *CFL* point of view, "morally" this condition is always violated because the information used to build the space derivative is always "lagging behind" the moving profile, as if it had an infinite speed.

NUMERICAL EXAMPLES ... SOME GRAPHS.

Looking back at ODEs, the news here are that, at least in the present sloppy context, a given time-advancing procedure can be "stable" or "unstable" *according to which spatial discretization method is coupled to.* Moreover, stability requirements may impose a relationship between the time and space discretization. In the case of the upwind scheme, stability requires $\alpha k < h$, i.e. for a given mesh size the time step must be small enough. This is a strong requirement on the numerics: if in a given computational box of size L we want more spatial resolution, i.e. more mesh points, we will have extra numerical work due to the fact that we need to reduce the time step as well.

If we now write the downwind approximation in implicit form

$$\frac{u_j^{n+1} - u_j^n}{k} + \alpha \frac{u_{j+1}^{n+1} - u_j^{n+1}}{h} = 0,$$

the one-step error is most conveniently calculated through the truncation error τ:

$$\tau^{n+1} = \frac{v_j^{n+1} - v_j^n}{k} + \alpha \frac{v_{j+1}^{n+1} - v_j^{n+1}}{k} \tag{5.44}$$

from which by Taylor expansion one obtains at the dominant order (we leave the proof to the reader)

$$e^{n+1} \sim -k\tau^{n+1} \sim -\frac{1}{2}\alpha k(h - \alpha k){v''}_j^n \tag{5.45}$$

which is just the opposite as (5.45). Strange enough, this seems to suggest that we have stability for $\alpha k > h$, i.e. for a *large enough* timestep. From the *CFL* point of view nevertheless this is not so strange: to be stable the method must get information on the side from which the profile is coming to build the space derivative, but it may even get information on the othes side if it looks at a sufficienly later time: in some sense, the method has lost its possibility of looking at the left at the same time, but catches it by looking on the right into the future. Nevertheless, this method has weird properties because you cannot decrease the timestep to increase the accuracy, so it is never used in practice[4]. This example is just there to tell that is not enough to switch from explicit to implicit schemes to gain stability automatically and safely!

Things go better if you take an implicit Euler with a *centered* approximation, i.e.

$$\frac{u_j^{n+1} - u_j^n}{k} + \alpha\frac{u_{j+1}^{n+1} - u_{j-1}^{n+1}}{2h} = 0$$

for which one can verify that the centered form for the space derivatives gives an error contribution to the space discretization which is *second order in h*, so if we limit ourselves to looking at the truncation error at the lowest possible order we get

$$e^{n+1} \sim -k\tau^{n+1} \sim \frac{1}{2}\alpha^2 k^2 {v''}_j^n \tag{5.46}$$

which following the former reasoning suggests that the scheme is always stable. It is in fact so, as we will see more rigorously in the following Section. Here adding stability when looking into the past for a small time step and stability when looking into the future for a large time step combines to give overall stability for any parameter values.

NUMERICAL EXAMPLES ... SOME GRAPHS.

5.6.2 Convergence and stability : is zero-stability for PDEs meaningful?

In the previous Section we saw how information on the truncation errors can give us some information on the possible stability or instability of the method. Here we will face the problem more formally. As for ODEs, convergence for initial-value problems described by a PDE is a requirement

[4]moreover, the linear system you obtain is not diagonally-dominant when p falls into the stability region, which gives additional troubles in solving it with standard routines.

concerning the *global error*, defined at a time step $n+1$ as the difference between the numerical solution and the true one:

$$E_j^{n+1} = u_j^{n+1} - v_j^{n+1} \tag{5.47}$$

where as usual we do not impose $u^n = v^n$: this is the error accumulated during the simulation, and not the error on just one step. It may even be $u_j^0 \neq v_j^0$ because of the machine rounding error on the initial condition (or for other strange reasons ... but nobody would be so crazy to force $u_j^0 \neq v_j^0$ beyond rounding errors and pretend obtaining a good solution afterwords!). As for the one-step error, one may take some norm, or consider the error pointwise.

While consistency relates to one-step and/or truncation errors, the true requirement, which really matters, turns out to be *convergence* on a given finite time $T = Nk$. As for ODEs, we say that the numerical method is *convergent* if:

$$||E^N|| \to 0 \;\text{ as } k, h \to 0, N \;\to \infty,\; T = Nk \;\text{ fixed} \tag{5.48}$$

i.e. the *global* error can be made as small as possible when reducing both time step and mesh sizes, but at a given finite time and letting the number of steps grow.

The problem is now to see under what conditions we can expect convergence of our scheme. We limit ourselves to the case of a linear PDE of the form $\partial_t v = F(v,t)$ where F is a linear operator containing space derivatives, and to a linear method of the form :

$$\boldsymbol{u}^{n+1} = (I+A)\boldsymbol{u}^n + \boldsymbol{c}(t_n, t_n+1) \tag{5.49}$$

which is just a rephrasing of (5.28) for a different A (I being the identity matrix and $A = A(k,h)$) that works for both explicit and implicit schemes. Proceeding as for ODEs we get :

$$\boldsymbol{v}^{n+1} = (I+A)\boldsymbol{v}^n + \boldsymbol{c} + k\boldsymbol{\tau}^{n+1}$$

whence

$$\begin{aligned} E^{n+1} &= \boldsymbol{u}^n + A\boldsymbol{u}^n + \boldsymbol{c} \\ &- \left[\boldsymbol{v}^n + A\boldsymbol{v}^n + \boldsymbol{c} + k\boldsymbol{\tau}^{n+1}\right] \\ &= \boldsymbol{u}^n - \boldsymbol{v}^n + A\left(\boldsymbol{u}^n - \boldsymbol{v}^n\right) - k\boldsymbol{\tau}^{n+1} \\ &= (I+A)\boldsymbol{E}^n - k\boldsymbol{\tau}^{n+1} \end{aligned}$$

So far it is the same story: at each step the global error is not only increased by the one-step error $\boldsymbol{e} \sim -k\boldsymbol{\tau}$, but amplified by the factor $(I+A)$. It is this amplification that makes some schemes good (stable) and some other bad (unstable).

Now for a given value of k we take the upper limit, $\tau(k,h)$, on the values of $\|\boldsymbol{\tau}^n\|$ for $n = 0, \cdots, N$ $(T = k\,N)$. We use $\tau(k,h) = \max_{n=0,\cdots,N} \{\|\boldsymbol{\tau}^n\|\}$ for instance, recalling that this upper limit was a function of k, and now depends on h as well. Then we take some norms and use the classical inequalities, so to have

$$||\boldsymbol{E}^{n+1}|| = ||I + A||||\boldsymbol{E}^n|| + k\tau(k,h).$$

Once iterated, we can write

$$||\boldsymbol{E}^n|| \leq ||I + A||^{(n)}||\boldsymbol{E}^0|| + k\tau(k,h)\sum_{m=1}^{n}||I + A||^{(n-m)} \tag{5.50}$$

where (sorry about that!!!) we introduced some weird notations: whenever a possible confusion in a formula arises between upper indices indicating time or exponents of some power, the latter (contrary to common sense) will be put into parentheses, so upper indices *with* parentheses will indicate powers while indices *without* parentheses will indicate times.

If now $||I + A||^{(n)}$ is uniformly bounded by some function $C(T)$ (i.e. constant if we keep the time fixed as we do when studying convergence), such that $||I + A||^{(n)} \leq C(T)\;\forall\, m, k, h$, we can establish an upper bound like

$$||\mathbf{E}^n|| \leq C(T)||\mathbf{E}^0|| + C(T)T||\boldsymbol{\tau}(k,h)||.$$

and taking the limits $k \to 0$ $(T = Nk$ finite$)$, $h \to 0$ convergence follows. This is the *Lax equivalence theorem* for linear initial-value PDEs: stability in the form $||I + A||^{(n)}$ bounded + consistency $\to$ convergence.

Otherwise no such upper bounds can be established and, apart from the term $||I + A||^{(n)}||\boldsymbol{E}^0||$ which comes purely from the rounding error and can possibly be discarded, we cannot guarantee that the truncation error part remains bounded as $k, h \to 0$, $N \to \infty$ even in the case of consistency (i.e. $\tau(k,h) \to 0$ at least as $\mathcal{O}(k) + \mathcal{O}(h)$).

Once we come to this conclusion, we realize that the notion of convergence is related to some form of *stability* exactly as for ODEs: i.e a requirement according to which *the global error does not get too much amplified at each time step.*

But if for ODEs, one-step and in general zero-stable methods allow such a function, *does such a function $C(T)$ exist for PDEs ?* As a starting step, we can diagonalize the matrix A and write the scheme (5.49) in the eigenvector basis for the operator A (we suppose we can do that : it is not guaranteed, but suppose we can). We recall that for an eigenvector basis $\{\boldsymbol{w}_{(l)}\}$ we have $A\boldsymbol{w}_{(l)} = \lambda_l \boldsymbol{w}_{(l)}$ where λ_l are eigenvalues and subscripts in parenthesis indicate an index running on the basis. Then when developing on this basis we write

$$\begin{aligned}
\boldsymbol{u}^n &= \sum_l U_{(l)}^n \boldsymbol{w}_{(l)} \\
A\boldsymbol{u}^n &= A\sum_l U_{(l)}^n \boldsymbol{w}_{(l)} = \sum_i U_{(l)}^n \lambda_{(l)} \boldsymbol{w}_{(l)} \\
\boldsymbol{c} &= \sum_l C_{(l)} \boldsymbol{w}_{(l)}
\end{aligned}$$

so the scheme (5.49)

$$\boldsymbol{u}^{n+1} = (I + A)\boldsymbol{u}^n + \boldsymbol{c}$$

rewrites $\forall\, l$ as

$$U_{(l)}^{n+1} = (1 + \lambda_{(l)})U_{(l)}^{n+1} + C_{(l)}$$

This is formally equal to (5.49) but we have a set of scalars instead of a vector and a linear operator. Nevertheless we can proceed as before, introduce a pointwise global error on the components $E_{(l)}^{n+1} = U_{(l)}^{n+1} - V_{(l)}^{n+1}$ and get $\forall\, l$:

$$|E_{(l)}^N| \le |1 + \lambda_{(l)}|^{(N)} |E_{(l)}^0| + k\tau(k,h) \sum_{m=1}^{N} |1 + \lambda_{(l)}|^{(N-m)} \tag{5.51}$$

(where E and τ are not the same as before, of course).

Let's take now a specific example for which we can calculate eigenvalues explicitly: the explicit Euler centered method for the advection equation with periodic b.c., which is exactly in the form (5.49) with $\boldsymbol{c} = 0$ and a $M \times M$ matrix A (related to (5.34))

$$\begin{pmatrix}
0 & -p/2 & 0 & \dots & 0 & 0 & 0 & \dots & 0 & p/2 \\
p/2 & 0 & -p/2 & \dots & 0 & 0 & 0 & \dots & 0 & 0 \\
 & & & & & \dots & & & & \\
0 & 0 & 0 & \dots & 0 & -p/2 & 0 & \dots & 0 & 0 \\
0 & 0 & 0 & \dots & p/2 & 0 & -p/2 & \dots & 0 & 0 \\
0 & 0 & 0 & \dots & 0 & p/2 & 0 & \dots & 0 & 0 \\
 & & & & & \dots & & & & \\
0 & 0 & 0 & \dots & 0 & 0 & 0 & \dots & 0 & -p/2 \\
-p/2 & 0 & 0 & \dots & 0 & 0 & 0 & \dots & p/2 & 0
\end{pmatrix}$$

where $p = \alpha k/h$. One can show that the eigenvalues of A are :

$$\lambda_{(l)} = i\frac{\alpha k}{h}\sin\left(\frac{2\pi}{L}lh\right) \equiv i\frac{k}{h}\beta_l\,, \quad l = 1\dots M\,, \quad -\alpha \le \beta_l \equiv \alpha\sin(2\pi lh) \le \alpha \tag{5.52}$$

We can now evaluate the amplification factor:

$$\begin{aligned}
|1+\lambda_{(l)}|^n &= \left|1+i\beta_l\frac{k}{h}\right|^n \\
&= \left(\left||1+i\beta_l\frac{k}{h}\right|^{1/k}\right)^{nk} \\
&= \left[\left(\left||1+i\beta_l\frac{k}{h}\right|^{h/k}\right)^{1/h}\right]^{t_n} .
\end{aligned}$$

If $k \to 0\,, h \to 0$ in such a way that $k/h \to$ const (and $t_n = nk$ is fixed) this quantity $\to \infty$: this because of the $1/h$ at the exponent! So the fact that for PDEs, contrarily to ODEs, we have amplification factors that contain explicit the mesh size h and can go to ∞ as $h \to 0$ do not ensures convergence for a simple explicit Euler method, which is consistent. By extension, for PDEs we cannot be sure that a consistent zero-stable method is convergent!

5.6.3 Absolute stability

For ODEs we obtained that the global error at a time $T = Nk$ is bounded by a function $C(T)$ ($C(T) = e^{KT}$ for one-step methods) times a term $\mathcal{O}(k)$ in such a way that it can go to 0 as $k \to 0$ (weak requirement, zero-stability). Zero-stability was a nice concept because associated to consistency it implied convergence. Nevertheless it was not all because in practice we needed the stronger requirement of *absolute stability*, i.e. that for a *finite chosen k* the error stays uniformly bounded at any time T (even $T \to \infty$) by a constant C (dependent on k).

We just saw that for PDEs we cannot guarantee that such a $C(T)$ exists, but still we can enforce absolute stability. Remember that absolute stability for ODEs has usually a meaning for exact solutions that are themselves bounded, because it is hard to pretend a forever-bounded error for a solution which has no bound and grows forever, like an exponential[5]. It turns out that for most of the PDEs used in practice, for some initial conditions the true solution *is* bounded, and it is enough to imagine that we are applying the scheme to these solutions : in most cases it will work fine even for unbounded solutions. An example is a stable scheme for calculating a linear instability... TO BE DEVELOPED LATER

With reference to expression(5.50), it is enough to require $||I + A|| \equiv$

[5]in this case we could ask for an error that grows slower than the solution, but it is still an active research subject and it goes a little bit too far.

$a(h,k) < 1$, since in this case :

$$\begin{aligned}
||\boldsymbol{E}^n|| &\leq a^{(n)}||\boldsymbol{E}^0|| + k\tau(k,h)\sum_{m=1}^{n} a^{(n-m)} \\
&= a^{(n)}||\boldsymbol{E}^0|| + k\tau(k,h)\frac{1-a^{(n)}}{1-a} \\
&\to k\tau(k,h)\frac{1}{1-a} \qquad \text{as } n\to\infty
\end{aligned}$$

and the error is the bounded by $C(k) = k\tau(k,h)\frac{1}{1-a(h,k)}$ which can be made small enough by choosing a convenient region in the plane (h,k). This will be the region of absolute stability for the method.

Equivalently, in an eigenvector basis it is enough to require $|1+\lambda_{(l)}| < 1$. In other words, an absolutely stable scheme will *contract* the solution, which will eventually be dissipated to 0, but this is the price to pay for avoiding catastrophic explosions.

It is interesting to examine the marginal case $||I+A|| \equiv a = 1$, for which $\sum_{m=1}^{n} a^{(n-m)} = n$ and then :

$$||\boldsymbol{E}^n|| \leq ||\boldsymbol{E}^0|| + kn\tau(k,h) = ||\boldsymbol{E}^0|| + T\tau(k,h)$$

and the error will grow at most linearly with time (this is formally the case in which at each step the one-step error at most adds at each step). Formally the region of stability includes the marginal case, i.e. the method is absolutely stable if $||I+A|| \leq 1$.

This is really nice, but nevertheless evaluating the eigenvalues of the matrix involved in the numerical method can be a complete nightmare. Luckily, there is a procedure known as Von Neumann stability analysis that for homogeneous PDEs gives an answer if we pretend to be in a periodic case.

5.6.4 The Von Neumann criterion for stability

Chapter 6

Implementation

In this chapter we have several objectives. We will present very briefly the `GNU` build system, or `autotools` which are the standard software engineering tools in `linux`. We will not enter into much details we just want to avoid to write Makefiles, to care about portability and source file dependencies to name a few common tasks, that's all.

We also want to provide the reader with the basics of `C++`. This is not a `C++` course and it would not be very reasonnable to pretend otherwise. We will indeed just concentrate on the `C` within `C++` and particularly just on a tiny subset of it. We will not dive into `C++` oddities not because we think this is useless[1] but because it takes years to master features of `C++` such as inheritance, polymorphism and template. Why `C++` then, and not directly the `C` language ? Because we do not want you to mess around with pointers ! Even if you are a `C` expert you must admit that a significant part of `C` programming is dedicated to debug memory problems. The `C++` lacks vectors and matrices classes specifically dedicated to numerical computations but we can use a popular library that can catch `fortran` performance in some cases : `blitz++`. So, a tiny subset of `C++` equipped with an array library such as `blitz++` is all we need to implement the numerical projects during this course.

Why using a compiled language ? We must admit that it is a common habit in computational astrophysics to code in compiled languages such as `fortran` or `C++`, very close to metal indeed. It is just not our habit to use some very famous interpreted languages [2] but using them is quite ok if you feel like it (again, it is simply not *our* habit).

So let me argue a bit in favour of compiled languages by telling a short anecdote. A colleague tried to convince me that using one of these famous interpreted softwares is the only way to computation by making the following analogy : prefering to use compiled languages instead of these softwares is

[1]Indeed, we just think the opposite !

[2]whose names we cannot pronounce because there are not *free* as in *free* speech ...

like entering a car and saying “This bike is useless, there are no pedals and handlebar”. Indeed, if I were not that slow for retorting I could have said (it occurred to me 2 minutes late actually) that prefering these softwares to compiled languages is like entering a formula 1 and say “This car is crap, there is no radio” ...

By the way, the whole idea is to code everything from scratch. Of course, in the professional world you might want to use very efficient libraries to solve i.e. linear systems of equations, Fast Fourier Transform, etc ... Ok, you can use these libraries as black boxes and if they do exist we strongly suggest that you do so *but* nonetheless our aim is again to code everything from scratch as far as we can get. We want to give you an hint on how these black boxes are made and not how to use them.

6.1 Autotools *inferno*

As a programmer there is a common evolution for taking care of the building and portabilities issues of your projects. First, you just do not want to care about thoses things because they are a loss of time and you think, “well, just a short `bash` script will do”, with only the compiler command line and its options into it. After messing around for a while and losing a lot of time, you convince yourself that trying to develop a decent `Makefile` that will handle for instance the dependencies between your source files is a better idea. Eventually, you will be faced with the problem to export your code to another computer and you will play somehow with portability issues which are kind of tricky. You try to write some `bash` script to automatised the whole job but you realise that indeed this job is a hard one and that unfortunately a `bash` script is far from being portable. When you have reached this third phase of your developer life it is time to use a descent [3] build system such as the `GNU` build system, aka the `autotools`. What we do propose here is to save you a lot of time by skipping the two first phases and dive into the `autotools` world right away.

If you are `linux` oriented you have probably already installed the so-called *autoconfiscated* packages as a simple user. They come in the form of a tarball, say `foobar-0.1.tar.gz`, and you install them in the following manner

```
> tar xvfz foobar-0.1.tar.gz
> cd foobar-0.1
> configure
> make
> sudo make install
```

[3]Well, yeah, as a matter of fact the use of this particular adjective could be debated ...

where `>` is the shell prompt. We will assume that "`.`" is in your path, otherwise you need to put `./` in front of every command or simply add `export PATH=.:$PATH`. in your `.bashrc` file.

From the user point of view one must admit that it is quite simple. You do not have to bother with the path to the necessary libraries for instance, the `configure` script does it for you. Well, at least if the developer did a good job.

Well, from the point of view of the developer the `autotools` are not that nice to use but still doing basic and standard stuffs is not that difficult. If you want to write `autoconf` macros for a new feature or library that you want to check for, well, it is a completely different story and everything becomes nightmarish. Fortunately, we will happily avoid this ! So please, do not be scared. We will use the strict minimum of the `autotools` capabilities. Remember that the main idea is to avoid to write Makefiles by hand and forget about file dependencies. As a supplementary benefice, your program is ready to be installed on different `UNIX` systems even those you probably never heard off.

Say that your project is called `foobar`. You must create a directory with that name with the following command

```
> mkdir foobar
```

When you are done with that, you need to create and edit some files. The first one can be considered to be the `autoconf` control file and is called `configure.ac` (where `.ac` stands for `autoconf`). Here comes a minimalist version of this file that you will certainly complete with other usefull macros in the next future for projects of your own.

```
1  AC_INIT([foobar],[0.1],[yourmail@unice.fr])
2
3  AM_INIT_AUTOMAKE([-Wall -Werror foreign dist-bzip2])
4
5  AC_PROG_CC
6  AC_PROG_CXX
7
8  AC_CONFIG_HEADERS([config.h])
9
10 AC_CONFIG_FILES([
11           Makefile
12           src/Makefile
13 ])
14 AC_OUTPUT
```

In line 1, You need to specify the packages name, the version number and the e-mail address for bug reports. Line 3 initialise the companion to `autoconf`, called `automake` whose job will be to create Makefile for you. Then, with the lines 5 and 6 we check for an available `C` and `C++` compilers.

autoconf does a very complicated job but in particular it will generate some usefull output files. For instance, it will generate a header files containing #define macros such as #define PACKAGE "hello" for instance that you might want to uses in your code. It comes in handy when you need some library that might not be installed and you have to deal with #ifdef directives. This is the purpose of line 8. Finally, the macro at line 10 instructs the build system to generate the Makefiles. The AC_OUTPUT macro actually tells autoconf to create the output files.

We need to give instructions to automake as well. For this purpose a file name Makefile.am (where .am stands for automake) must be created. In our minimalist approach it specifies only that we will have a subdirectory src

```
SUBDIRS = src
```

that is all.

We must create the subdirectory src that will contain our source code with

```
> mdkir src
> cd src
```

the last line places us within this subdirectory where we have to add another Makefile.am containing

```
bin_PROGRAMS = foobar
foobar_SOURCES = foobar.cpp
```

The first line instructs automake that we have one executable (expressed by the prefix bin_) file named foobar. The second specifies the sources for this program. Ok, here it is very simple since we have just one source file. However the “beauty” of the thing is that with a lot of sources files (including .h and .cpp files, etc …) you do not have to worry about dependencies the build system deals with all that by itself.

We are left with the creation of our first program in C++, the classical “Hello world” example. Fire-up your favourite text editor to create and edit the foobar.cpp file with for instance

```
#include <iostream>
using namespace std;

int main() {

  cout << "Hello world !" << endl;

  return 0;
}
```

that should be located in the `src` directory. Do not worry, we will get back to what this code actually does in the next section about `C++`.

We are not through yet because we need to invoke `autoconf`, `automake` and related programs (`aclocal` and `autoheader`). Personally, I cannot remember the exact sequence and what does what. The good news is that I am not alone and this is the reason why the magical command `autoreconf` does everything right for us. The first time it has to be invoked with the following option to generate all the necessary files

```
> autoreconf --install
```

which prints out

```
configure.ac:8: installing './install-sh'
configure.ac:8: installing './missing'
src/Makefile.am: installing './depcomp'
```

If you look at what is now in the directory you will notice the creation of the `configure` script and the `Makefiles`. If you are curious enough to look within this files you will notice that they are kind of cryptic even if you are familiar with shell programming and the `make` program. Remember it is generated in the idea of being portable on any kind of Unix system and the shell script `configure` only use features common to every shell and not just you favourite one. It also follows the `GNU` project coding standards.

We are almost done. We just need to invoke the configuration script and then `make` like that

```
> configure
> make
```

That's it. You can run the `foobar` "Hello World" program just as usual

```
> src/foobar
```

to get the expected results

```
Hello world !
```

Now, if you want to install this very useful greeting program on your computer you can do it with

```
> sudo make install
```

Yeah, you need to be root because by default, following the `Unix` and `GNU` standards, the program will be installed in `/usr/local/bin`. Ok, you will not be root on the university computers for instance, in that case you can always specify where you prefer to install everything whith the configuration script called in the following manner

```
> configure --prefix=/path/to/your/favourite/folder
```

and then simply

```
> make install
```

without the `sudo`.

If now you think that it is the best program ever written and that you want to distribute it to the masses here goes what you need to do

```
> make distcheck
```

You obtain too wonderful standard `Unix` package `foobar-0.1.tar.gz` and `foobar-0.1.tar.bz2`. It contains everything needed to build your program on any kind of `Unix` system[4]. By having a look at the beginning of this section you know how to handle this `autoconfiscated` package.

6.2 A touch of C++

Again, our purpose is not to give a lecture about `C++`. We will not cover the most advanced features here but concentrate on what we need only. We will try to learn trough a set of selected examples rather than exposing the language features one after the other. If you do not have any background in computer programming, you can expect this section to be kind of difficult to swallow ...

If you really want to learn `C++` you can read Eckel et al. (2000, 2003) but you must know the `C` language first. This is not for the lighthearted: we are not talking about hundredth of pages but thousands of pages *though* very well written pages I think. These books do not only teach you the language but how to program in `C++`, which is quite a different business: you can know the syntax perfectly and write some very bad code. Then, if you want to go even further, Gregoire et al. (2011) is a good reference to go for. Remember that *it's a long way to the top if you wanna rock'n'roll* [5].

6.2.1 Hello world

In the previous section we already introduced the classical "Hello world !" program. Let's look at it in details now. The first line with "`#include <iostream>`" is not `C++` actually but a `C` pre-processor macro (`cpp` below). It includes the content of a file. In `C++` things must be *declared* before they are used. These declarations are put together in *header* files usually ending in `.h`. For instance to include the content of the header file `foo.h` you will use `#include "foo.h"`. If the header file to be included is a standard one,

[4]Well, for such a simple program it is true. For complicated project some more work might be necessary though.

[5]Ridin' down the highway, Goin' to a show, Stop in all the by-ways, Playin' rock 'n' roll ... :-)

i.e. from the standard library, you would use `<>` instead of `""`. If you are a `C` programmer there are some slight changes. In `C` to include the `math.h` file you would have use `#include <math.h>` but in `C++` you must use a slightly different syntax with `#include <cmath>`. You skip the `.h` and you add a `c`, no big deal.

So, `#include <iostream>` includes all the necessary declaration for the input/output stream. Streams are used in `C++` for input/ouput operations like printing the message "Hello world !" on the standard output.

Then `using namespace std;` does exactly what it says : we use the namespace called "std". A namespace contains names and has its importance with large project because you might run out of names after a while. For instance, you might want to declare a `flag` variable in many places and you must have to face conflicts of declarations. You can have your *different* variables `flag` in different namespaces to solve this problem. The point is that the declarations of the Standard Template Library (STL), part of the `C++` standard, are embedded in the "std" namespace.

As in `C`, your program must have a function called `main` that returns an `int` value. The main function can have some parameters. In `linux` you can read the command line argument but we just skip that for the moment. A function in `C++` is a programing entity that will do some program, to which we pass some parameters and that can return a "value". Here the value is an `int` that stands for (signed) integers. Within a function you must put some declaration sand statements to be executed. They are grouped in a scope limiting the core of the function definition. A scope starts by a `{` and end with `}`. `main()` states that the main function takes no argument. You can write `{` just after `main()` or in a new line or whatever because the source code can be formated as you like. Some like to start their `{` right after `main()`. Others prefers to go to the next line. You won't believe it but there is a lot of debate about this behaviour. Do whatever you like, but be consistent. The readability of your code is a *very important* matter.

Then comes the statement

```
    cout << "Hello world !" << endl;
```

"`cout`" is the stream corresponding to the standard outpout. To sent something to this stream you must use the following operator `<<`. It will have the effect of printing things out on your terminal. `endl`, a manipulator of the stream, has the effect of inserting a new line character.

Since we are finished with our program we must end the `main` function by returning `0` that means everything went fine. We do this with the following statement : `return 0;`.

6.2.2 Computing the square root

We are going to write a small program that ask for a number and gives its square root. Ok, it is not a very interesting program but it illustrates and introduces some features that we need.

First of all, we need the math library to compute the square root that defines the `sqrt` function. We need to instruct the `autotools` to link with this library by adding this macro to the `configure.ac` file just after the call to AC_PROG_CXX

```
AC_CHECK_LIB([m],[sqrt],[],
  AC_MSG_ERROR([The math library must be installed]))
```

This `autoconf` macro checks the presence of a function (argument 2) in a given library (argument 1). I am afraid that the brackets must be present. In some situations it does not matter in other it does since one never remember why it is so, just put the bracket and everything is going to be just fine. Argument 3 instructs `autoconf` what to do if we find the library. In our case, we want the default behaviour and we let the `autotools` set the linker and `cpp` flags for us. Then, the last argument is the behaviour in case we do not find the library. We use a macro dedicated to print out an error message. We do not give options to the user in that case: the presence of the math library is mandatory. In some situations, it is not the best thing to do and more complicated things have to be done. For the time being it is quite sufficient.

Now it is time to create and edit a new file that will contain the source code for our programm `asksqrt.cpp` and that is located in the `src` subdirectory. We must tell to `automake` that we have a new `bin` file by modifying the `src/Makefile.am` like that

```
bin_PROGRAMS=hello asksqrt
hello_SOURCES=hello.cpp
asksqrt_SOURCES=asksqrt.cpp
```

remembering that we already have our "Hello World !" program. Now, let's create `asksqrt.cpp` with

```
#include <iostream>
#include <cmath>

using namespace std;

int main() {

  double x;
  cout << "Enter a number: ";
  cin >> x;
  cout << "The square root of " << x
```

```
        << " is " << sqrt(x) << endl;
  return 0;
}
```

We declare a variable name `x` as a `double` which is a built-in `C++` type that stands for double precision floating point numbers. Note that in `C++` you can define your own types too. "built-in" type means that it comes with the language itself. By the way, the declaration can be put everywhere you like in the code, not necessarily at the beginning as in `C` or `fortran` with the restriction that something must be declared *before* it used. Of course, you have other built-in type such as `float` for single precision arithmetic [6], `int` for integer for instance.

`cin` is the input stream. You will have to enter the number whose square root you want to compute in the terminal. Then, the stream operator `>>` will put data from the stream into you `double` variable. You do not have to specify the type of the variable.

The other line in the code just prints out a message and the results of the computation. If you want to test the program, you need to call `make` in the main directory of our project `foobar` where `configure.ac` is located. You should get first the following line

```
Enter a number:
```

to which you can reply `27`

```
Enter a number: 27.
```

and press return to obtain

```
The square root of 27 is 5.19615
```

Now, what if the user of this amazing program enter a negative number ? With `-1.` we obtain

```
The square root of -1 is -nan
```

a nan (not a number) exception from the IEEE standard of course !

We can actually test for the positivity of `x` with an `if` statement by modifying our small program like that

```
  double x;
  cout << "Enter a number: ";
  cin >> x;
  if(x>=0.)
    cout << "The square root of " << x
         << " is " << sqrt(x) << endl;
```

[6] Again, the `C++` standard just requires that the size of a `double` is larger or equal to that of `float`.

```
  else
    cout << "Error: negative number." << endl;
```

The expression `x>=0.` is a boolean and is `true` or `false` according to the value of `x`. `>=` is a comparison operator but there are other like: `>`, `<=`, `<` self explained. If you want to test for equality the operator is `==`; for inequality it is `!=` meaning *different from*.

Be extremely careful here: `==` not just one `=` which is the assignment operator ! You can use `=` in an `if statement` like that

```
  if(x=0.) doSomething(); // Legal but dangerous !
```

it is legal but very dangerous and have a complete different meaning that one might expect. `x` is set to 0. And the statement `x=0.` has a value, the value of `x` actually, so this `if` statement eats a `0` which means `false` and the function `doSomething` will never be called. Usually, compilers warn you for this kind of obscure programming habits.

6.2.3 Playing with arrays

Since we are going to do some numerical computations we will play with vectors and matrices for sure. We then need arrays of floating point number to do our calculations. We do not necessarily know the size of a vector or a matrix before starting the computation because it might depend on other parameters given by the user of the program (e.g. the precision of the computation). Consequently, we need allocatable arrays. Of course, like in `C` you can allocate memory at your ease with dedicated operators (`new` and `delete`). If you want to spend, say, 90 % of your coding time by debugging pointer problems all well and good: we will not follow your path.

The bad news is that there is simply no allocatable array class in the `C++` standard library. Well there is a `vector` class actually, but no multi-dimensional arrays in `C++`. You can do such thing as vector of vectors (etc . . .) and so you could create your array classes in that way. For numerical computation it might not be very efficient though because your “matrix” might well not be stored in a contiguous block of memory.

We could develop an array class, in particular by overloading the definition of operator like `+`, `*` etc to obtain something quite descent (one of the author does just that) at low cost. We can instead use libraries to do the job knowing that we will not use all the capabilities of the library. There are two good possibilities :

- the boost library (`www.boost.org/`) and its `array` class. However, its use can be kind of cryptic, in particular initialization of array data.
- The `blitz++` library. It is a very powerful template library with a lot of features you can find in interpreted languages with implicit looping and

even the notion of stencils (useful for finite differencing techniques). It is included in `linux` distribution such as debian or ubuntu. So our choice is made. However, there is something quite strange. When we started to write these section (June 2012) the main library web page `www.oonumerics.org/blitz/` expired ! We can still found everything, including the source code, at `blitz.sourceforge.net` and `http://sourceforge.net/projects/blitz/`. The manual is well written so if you think you might need more advanced features you can have a look into it.

What we need to do is to tell the `autotools` to check for the `blitz++` library. We can do it by adding the following line to our `configure.ac` file

```
AC_CHECK_LIB([blitz],[__cxa_atexit],[],
 AC_MSG_ERROR([The blitz++ library must be installed]))
```

This gives the false impression that it is always that easy with `autoconf`. Unfortunately, this is simply not true. With `C` libraries indeed it can be that easy. With `C++` it is a different story. The `AC_CHECK_LIB` macro test by compiling a little program that we can link and use the library correctly. For this, it needs to be provided by a function. However, for a pure `C++` library there is no function definitions. There are only classes. To take this into account some modyfication to the `AC_CHECK_LIB` macro is sufficient but clearly out of the scope of these lectures. With `blitz++`, a pure `C++` class, we do not have any standalone functions (but methods in classes of course !). However, we are still lucky because we still have symbols to provide to the macro

`__cxa_atexit` is such a symbol as you can find by inspecting the library content (with the `ar -t` and `nm` unix commands). Another possibility would be to use the symbol `_blitz_id` as mentioned on `http://sourceforge.net` `blitz++` web page. However, for recent version of the library this symbol no longer exists.

Is there a better way to proceed ? The answer is yes. Someone develops a macro for checking for the library. You can find it by searching for `ac_cxx_lib_blitz.m4` on the web but then we need to modify our `configure.ac` and `Makefile.am` files accordingly and the macro itself needs to be patched somehow. So, we check for the library with the method proposed above. It is not the best way to proceed but still it is easy and it works. Another possibility would be to hack the `autoconf` macro `AC_CHECK_LIB` and adapt it to suit our needs. This goes out of the scope of these course.

Let's create a new file arrays.cpp to play with the array classes. First we need to include the header file like this

```
#include <blitz/array.h>
```

and use the `blitz` name space like that

```
using namespace blitz;
```

Now in the core of our program we can defined a vector of `double` as easy as this

```
Array<double,1> v(4);
```

which is of size 4.

To initialize this vector you can assigned its values with simply

```
v=1.,2.,3.,4.;
```

Note that this way of initializing the vector is not something which is defined by `C++` ! The author of the library obtained this syntactic behaviour by overloading the “**,**” operator ! Yes, you can do this in `C++` . . .

You can send the arrays to some data streams to print them out for instance like that

```
cout << v << endl;
```

like for a `double`.

Of course, you can create arrays of any type for which arithmetic operations make sense such as Array<float,1> v; or Array<int,1> v;. Note that you do not need to specify the size of the array when declared.

For a matrice, you just need to declare an array of rank 2 with the following obvious syntax

```
Arrays<double,2> m(2,2);
```

which declare a 2×2 matrix. Again you can initialize it with

```
  m=1.,2.,
     3.,4.;
```

and you can do things like that

```
Array<double,2> m2;
m2=sqrt(m);
Array<double,2> m3;
m3=m+m2;
```

etc . . .

6.2.4 Writing results into a file

6.2.5 And now, what ?

These notes about `C++` might not be sufficient to develop an advanced program. Again, the purpose is not to learn `C++` but how to integrate differential equations numerically. We learn a few basic things and how to develop a small program and compile it. This was all we needed to start coding by ourselves. In Chapter 7 we will face some real numerical problems. When we will be in need of some `C++` news features or concepts, we will explain it *in situ*.

Chapter 7

Projects

Now, it is time for a little action. Well, you know "action", like spending hours in front of the computer, even talking to it, perhaps insulting it and, of course, drinking too much coffee . . . We will focus our considerations on three types of PDEs: diffusion, advection and wave equation. From a mathematical point of view (see chapter 5) they are of a different nature corresponding to parabolic (the first one) and hyperbolic (the others) PDEs (the third foundamental kind, elliptic equations, is so far missing from these lectures). Numerically we will see that the treatment is somehow different. However, from a physical point of view we might say that all arises because of the existence of some *conserved* quantity. Let's consider a scalar quantity v in one dimension. The conservation of this quantity takes the following form

$$\partial_t v + \partial_x F = 0 \ , \tag{7.1}$$

where F is the flux of the quantity. F might depend on v or its gradient $\partial_x v$, etc . . .

Now, if the flux is simply proportional to v, for instance $F = c\,v$, c being a constant, we obtain the advection equation

$$\partial_t v + c\,\partial_x v = 0 \ . \tag{7.2}$$

If the flux is proportional to the gradient, $F = -D\,\partial_x v$ we obtain the diffusion equation:

$$\partial_t v - D\,\partial_x^2 v = 0 \ . \tag{7.3}$$

The conserved quantity could also be a vector $\boldsymbol{v} = \left(v^1, v^2\right)$ which leads to the following conservation equation

$$\partial_t \boldsymbol{v} + \partial_v \boldsymbol{F} = 0 \ . \tag{7.4}$$

As for the advection equation we choose the flux vector to be proportional to $\boldsymbol{v}$: $\boldsymbol{F} = A\,v$ where A now is a matrix rather than a scalar. Taking the matrix A to be

$$A = \begin{pmatrix} 0 & c \\ -c & 0 \end{pmatrix} , \tag{7.5}$$

where c is a constant, we obtain the following system of equations for the component of $\boldsymbol{v}$

$$\begin{cases} \partial_t v^1 + c\,\partial_x v^2 = 0 \\ \partial_t v^2 - c\,\partial_x v^1 = 0 \end{cases} . \tag{7.6}$$

By combining both equation, e.g. deriving the first with respect to t, using the fact that ∂_t and ∂_x commute and, inserting the expression of $\partial_t v^2$, we obtain the wave equation

$$\partial_t^2 v^1 = c^2\,\partial_x^2 v^1 , \tag{7.7}$$

for the first component of v^1. A similar equation holds for the second component v^2. Note that in this form, we recognize c as the propagation speed of the wave.

7.1 The diffusion equation

7.1.1 1D diffusion equation

We want to determine a numerical approximation to the solution $v(x,t)$ of the 1D diffusion equation given by

$$\partial_t v(x,t) - D\,\partial_x^2 v(x,t) = 0 , \tag{7.8}$$

where D, the diffusion coefficient, is assumed to be constant on the whole integration domain $x \in [a,b]$.

The above equation must be supplemented by boundary conditions in $x = a$ and $x = b$ since the derivation respect to x is of order 2. We will use *Dirichlet* boundary conditions by specifying the values of the function at the boundaries

$$\begin{aligned} v(a,t) &= \gamma_a(t) \qquad \text{and} \\ v(b,t) &= \gamma_b(t) , \end{aligned} \tag{7.9}$$

where γ_a and γ_b are two given functions of time.

To fully specify the problem there remains to provide the initial condition with

$$v(x,t_0) = \overset{\circ}{v}(x) , \tag{7.10}$$

where t_0 is the initial instant and $\overset{\circ}{v}(x)$ is a given function of space defined on $[a,b]$. For evident reasons of compatibility, it must be $\overset{\circ}{v}(a) = \gamma_a(0)$ and $\overset{\circ}{v}(b) = \gamma_b(0)$, while when plugging both initial and boundary conditions into the equation itself, and evaluating it at $x = a$ or $x = b$, and $t = 0$ one derives further compatibility conditions:

$$\begin{aligned} \dot{\gamma}_a(0) &= D \overset{\circ}{v}{}''(a) \qquad \text{and} \\ \dot{\gamma}_b(0) &= D \overset{\circ}{v}{}''(b) \ , \end{aligned} \tag{7.11}$$

primes and dots indicating respectively derivatives with respect to space and time.

1- We will first study the numerical scheme called "Forward Time Centered Space" (FTCS) defined by

$$\frac{u_i^{n+1} - u_i^n}{k} = D \, \frac{u_{i+1}^n - 2\, u_i^n + u_{i-1}^n}{h^2} \ , \tag{7.12}$$

where u_i^n represent the numerical approximation to $v(x_i, t_n)$, h and k are respectively the spatial and temporal steps with $x_i = i\,h + a$ and $t_n = t_0 + n\,k$ with $h = \frac{b-a}{N-1}$, for $j = 0, \cdots, N-1$ and $n \geq 0$.

1-a) What is the order of the FTCS method ? Compute the dominant term in the local truncation error τ.

1-b) Substituting a *Fourier* mode $\xi_l^n \, \mathrm{e}^{j\,k_l\,x}$ in Eq. (7.12), where $j = \sqrt{-1}$, prove that the amplification factor $\left| \frac{\xi_l^{n+1}}{\xi_l^n} \right|$ is given by

$$\left| \frac{\xi_l^{n+1}}{\xi_l^n} \right| = \left| 1 - 4\,\alpha \sin^2\left(\frac{h\,k_l}{2} \right) \right| \ , \tag{7.13}$$

where the *Courant* number is defined by $\alpha = \frac{k\,D}{h^2}$.

1-c) By requiring the condition $\left| \frac{\xi_l^{n+1}}{\xi_l^n} \right| < 1$, deduce that this scheme is stable in the *Von Neumann* sense under the following condition

$$k < \frac{1}{2} \frac{h^2}{D} \ . \tag{7.14}$$

1-d) What is the main drawback imposed by the condition (7.14) ?

2- Write a program to solve the diffusion equation with

$$\begin{aligned}
\overset{\circ}{v}(x) &= \mathrm{e}^{-\frac{1}{4}\frac{x^2}{D\,t_0}} \ , \\
\gamma_a(t) &= \sqrt{\frac{t_0}{t}}\,\mathrm{e}^{-\frac{1}{4}\frac{a^2}{D\,t}} \ , \\
\gamma_b(t) &= \sqrt{\frac{t_0}{t}}\,\mathrm{e}^{-\frac{1}{4}\frac{b^2}{D\,t}} \ ,
\end{aligned}$$

where you can use $a = -b$ to simplify somehow.

2-a) Compare the following exact solution

$$v(x,t) = \sqrt{\frac{t_0}{t}}\,\mathrm{e}^{-\frac{1}{4}\frac{x^2}{D\,t}} \ ,$$

to the numerical results with the FTCS scheme. You might have recognized the *Green* function of the diffusion equation.

2-b) Choose a time step k in order to generate instabilities in the FTCS scheme.

3- We study now an *implicite* scheme "Backward Time Centered Space" (BTCS) given by

$$\frac{u_i^{n+1} - u_i^n}{k} = D\,\frac{u_{i+1}^{n+1} - 2\,u_i^{n+1} + u_{i-1}^{n+1}}{h^2} \ , \tag{7.15}$$

3-a) Show that this scheme is unconditionally stable.

3-b) The scheme being *implicit*, u_i^{n+1} is obtained by solving a linear system that you will write explicitely.

3-c) Write a program that solve this linear system using for e.g. the *Gauss-Seidel* method and integrate again the diffusion equation with the BTCS scheme with the same boundary conditions.

3-d) Verify the unconditional stability of the BTCS scheme. What are the pros and cons of implicit method with respect to explicit ones ?

4- We now consider the *Crank-Nicolson* scheme

$$\frac{u_i^{n+1} - u_i^n}{k} = \frac{1}{2}D\left(\frac{u_{i+1}^{n} - 2\,u_i^{n} + u_{i-1}^{n}}{h^2} + \frac{u_{i+1}^{n+1} - 2\,u_i^{n+1} + u_{i-1}^{n+1}}{h^2}\right) \ . \tag{7.16}$$

4-a) What is the order of this method ?

4-b) Study the stability of the *Crank-Nicolson* method.

4-c) Adapt your previous program to solve the diffusion equation with this numerical scheme.

7.1.2 2D diffusion equation

If we want to go to 2D, or even 3D, there is indeed not much difficulties since what we have done in 1D can be generalized to 2D without much work. Their is an additional dependence of $v(x, y, t)$ on y so our numerical approximation must now be indexed by a new index j to become $u_{i,j}^N$, where $i = 0, \cdots, N_x - 1$ and $j = 0, \cdots, N_y - 1$ where we introduced the total numbre of x and y values respectively N_x and N_y.

For instance, we can generalize the FTCS explicit scheme like this

$$\frac{u_{i,j}^{n+1} - u_{i,j}^n}{k} = D \left(\frac{u_{i+1,j}^n - 2\,u_{i,j}^n + u_{i-1,j}^n}{h_x^2} + \frac{u_{i,j+1}^n - 2\,u_{i,j}^n + u_{i,j-1}^n}{h_y^2} \right) , \tag{7.17}$$

were h_x and h_y are the spatial steps in x and y respectively. The domain of $\mathbb{R}^2$ we are interested in is the Cartesian product $[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$ defining h_x and h_y as

$$h_x = \frac{x_{\max} - x_{\min}}{N_x - 1} \tag{7.18}$$

$$h_y = \frac{y_{\max} - y_{\min}}{N_y - 1} \tag{7.19}$$

so that we also have

$$x_i = i\,h_x + x_{\min} \tag{7.20}$$

$$y_j = j\,h_y + y_{\min} \; . \tag{7.21}$$

5- Give the expressions for the BTCS and the *Crank-Nicolson* schemes in 2D.

6- For the *Von Neumann* stability analysis we now have to insert *Fourier* modes given by $\xi_{l,m}^n \, \mathrm{e}^{j\,(k_l\,x + k_m\,y)}$. Give the conditions for the stability of the three numerical schemes by showing that

- the BTCS and *Crank-Nicolson* are, as in 1D, unconditionally stable,
- the FTCS scheme is stable under the following condition

$$k < \frac{1}{2}\,\frac{h^2}{D} \; , \tag{7.22}$$

where h is defined by

$$h = \frac{h_x\, h_y}{\sqrt{h_x^2 + h_y^2}} \ . \tag{7.23}$$

7- Write a computer program that solve the 2D diffusion equation with *Dirichlet* boundary conditions and and initial profile $\overset{\circ}{v}(x, y)$ of your choice [1].

7.2 The wave equation

7.2.1 1D equation

We seek a numerical approximation to the solution $v(x, t)$ of the 1D wave equation

$$\partial_t^2 v(x, t) - c^2\, \partial_x^2 v(x, t) = 0 \ , \tag{7.24}$$

where c, the propagation speed, is assumed to be constant over the integration domain $x \in [a, b]$.

This equation could be used to describe for instance the vibration of a string. In this case, $v(x, t)$ would represent the displacement of the string at x for a given time t respect to the equilibrium situation.

We assume *Dirichlet* boundary conditions in a and b

$$\begin{aligned} v(a, t) &= \gamma_a(t) \\ v(b, t) &= \gamma_b(t) \ , \end{aligned} \tag{7.25}$$

where γ_a and γ_b are two given functions of time. Since the problem is order 2 in time we also need two initial conditions like these

$$\begin{aligned} v(x, t_0) &= \overset{\circ}{v}(x) \\ \partial_t v(x, t_0) &= w(x) \end{aligned} \tag{7.26}$$

where $\overset{\circ}{v}(x)$, $w(x)$ are two known functions of the spatial coordinates x and t_0 the initial instant. Without loss of generality we will assume $t_0 = 0$ below.

As usual, we define the spatial mesh with $x_i = h\, i + a$ for $h = \frac{b-a}{N_x - 1}$ and $i = 0, \cdots, N_x - 1$. The numerical approximation is sought over discrete values of time $t_n = t_0 + n\, k$ where $k > 0$ is the integration time step. For the numerical approximation to the solution of Eq. (7.24) the following notation will be used : $u_i^n \approx v(x_i, t_n)$.

1- We propose to integrate numerically Eq. (7.24) using central finite differences for the second order time derivative

$$\partial_t^2 v(x_i, t_n) \approx \frac{u_i^{n+1} - 2u_i^n + u_i^{n-1}}{k^2} \ , \tag{7.27}$$

[1] You can put some oscillating boundary conditions with cos and sin of different pulsations to get a psychedelic effect . . .

as well as the second order spatial derivative

$$\partial_x^2 u(x_i,t_n) \approx \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{h^2} \; . \tag{7.28}$$

1-a) Give an expression for the dominant terms in the one-step errors in t and x for the following numerical scheme obtained by using Eq. (7.27) and (7.28)

$$\frac{u_i^{n+1} - 2\,u_i^n + u_i^{n-1}}{k^2} = c^2\,\frac{u_{i+1}^n - 2\,u_i^n + u_{i-1}^n}{h^2} \; , \tag{7.29}$$

where the *Taylor* expansions will be done around (x_i,t_n).

1-b) Is this numerical scheme consistent ?

2- By inserting *Fourier* modes $\xi_l^n\,\mathrm{e}^{j\,k_l\,x}$ in Eq. (7.29), show that we obtain the following linear recurrence relation

$$\xi_l^{n+1} - (2-\gamma^2)\,\xi_l^n + \xi_l^{n-1} = 0 \; , \tag{7.30}$$

where $\gamma = \dfrac{2\,c\,k}{h}\,\sin\left(\dfrac{k_l\,h}{2}\right)$.

3- In order to obtain the solution to the linear recurrence relation Eq. (7.30) we seek solution of the form λ^n i.e. λ *to the power of* n (and not λ at time step n).

3-a) Prove that λ verifies the following characteristic equation

$$\lambda^2 - \left(2-\gamma^2\right)\,\lambda + 1 = 0 \; . \tag{7.31}$$

3-b) Show that when the discriminant Δ of Eq. (7.31) defined by $\Delta = \gamma^2\,\left(\gamma^2 - 4\right)$ is positive we have two roots $\lambda_\pm$ given by

$$\lambda_\pm = \frac{2-\gamma^2 \pm \gamma\sqrt{\gamma^2-4}}{2} \; , \tag{7.32}$$

with $|\lambda_-| > 2$.

3-c) When $\Delta < 0$, prove that both complex roots given by

$$\lambda_\pm = \frac{2-\gamma^2 \pm i\,\gamma\sqrt{4-\gamma^2}}{2} \; , \tag{7.33}$$

verify $|\lambda_\pm| = 1$.

4- According to the stability theory of numerical integration methods of ODEs,

4-a) Deduce from the above consideration that the condition for the stability of the numerical scheme Eq. (7.29) is obtained if the *Courant-Friedrich-Levy* criterion is fulfilled

$$\frac{c\,k}{h} < 1 \ . \tag{7.34}$$

4-b) Since c is the wave propagation speed could we expect intuitively such a result ?

5- The explicit scheme Eq. (7.29) is a two (time) step method. We do have the value of u_i^0 as initial condition given by $u_i^0 = \overset{\circ}{v}(x_i)$. However, u_i^1 is missing and must be derived from the condition $\partial_t v(x, t_0) = w(x)$. We can use the following little trick: we condiser our numerical approximation a step backward in time from t_0 : u_i^{-1},

5-a) Prove that the u_i^{-1}'s must verify

$$\frac{u_i^1 - u_i^{-1}}{2\,k} = w_i \ , \tag{7.35}$$

in order to preserve the spatial order of the numerical scheme, with $w_i = w(x_i)$.

5-b) Then, show that we can use the following relation to start our numerical scheme out

$$u_i^1 = \left(1 - \alpha^2\right)\, u_i^0 + \frac{\alpha^2}{2}\, \left(u_{i+1}^0 + u_{i-1}^0\right) + k\, w_i \ , \tag{7.36}$$

where $\alpha = \frac{c\,k}{h}$ is the *Courant* number.

6- Develop a computer program that implements this numerical method with the following simple boundary conditions : $w(x) = 0 \quad \forall x \in [a, b]$.

7.2.2 2D equation

The generalisation to 2D is again straightforward for this implicit scheme following the path presented in Sect. (7.1.2)

7- Give the 2D version of the numerical scheme described by Eq. (7.29).

8- Show that the *Courant-Friedrich-Levy* criterion can still be written in the form of Eq. (7.34) provided that h should be replaced by $h = \frac{h_x\, h_y}{\sqrt{h_x^2+h_y^2}}$ as in Eq. (7.23).

9- Generalise your 1D program to solve the wave equations in 2D.

7.3 The advection equation

7.3.1 1D equation

We consider the 1D advection equation for a real function $v(x,t)$ of the position x and time t given by

$$\partial_t\, v(x,t) + c\,\partial_x\, v(x,t) = 0 \ , \tag{7.37}$$

where the speed $c > 0$ is assumed to be a constant and where $x \in [0, L]$ with $L > 0$ and $t \geq t_0$. We further assume the following *periodic* boundary conditions

$$v(0,t) = v(L,t) \qquad \text{for all } t \geq t_0 \ . \tag{7.38}$$

We must also provide the initial profile $\overset{\circ}{v}(x)$ that will be advected

$$v(x,t_0) = \overset{\circ}{v}(x) \ , \tag{7.39}$$

where $\overset{\circ}{v}(x)$ is a given function of space defined on $[0, L]$.

As usual, the numerical approximations to the solution of Eq. (7.37) will be otained for $x_i = h\,i$ with the spatial step given by $h = \frac{L}{N_x-1}$ and $i = 0, \cdots, N_x - 1$. We also have $t = t_0 + n\,k$ where $k > 0$ is the time step and $n = 0, \cdots, \infty$. We will note these numerical approximations by u_i^n.

1- It is quite instructive to find the general form of the solution to Eq. (7.37). We introduce a new set of variables defined by

$$\begin{aligned} \varphi &= x - c\,t \qquad \text{and} & (7.40)\\ \mu &= x + c\,t \ . & (7.41) \end{aligned}$$

1-a) Show that Eq. (7.37) for these variables reduces to the following equation

$$\partial_\mu\, v(\mu,\varphi) = 0 \ . \tag{7.42}$$

1-b) Prove that the solution to Eq. (7.37) is of this form $v(x,t) = \tilde{v}(x - c\,t)$ where $\tilde{v}$ is a function of a real variable.

1-c) Finally, show that the solution to Eq. (7.37) is given by

$$v(x,t) = \overset{\circ}{v}\left(x - c\,(t - t_0)\right) \ , \tag{7.43}$$

where we recall that $\overset{\circ}{v}$ is the initial profile.

2- We make a first try at guessing a method of numerical integration of Eq. (7.37) like this:

$$\frac{u_i^{n+1} - u_i^n}{k} + c\,\frac{u_{i+1}^n - u_{i-1}^n}{2\,h} = 0\ . \tag{7.44}$$

2-a) Inserting the exact solution $v(x,t)$ in Eq. (7.44) determine the dominant terms in k and h for the truncation error. Show that the method is consistent.

2-b) Proceed to the *von Neumann* stability analysis by inserting a *Fourier* mode $\xi_l^n\, \mathrm{e}^{j\,k_l\,x}$ in the scheme Eq. (7.44) and show that the amplification factor $\frac{\xi_l^{n+1}}{\xi_l^n}$ is given by

$$\frac{\xi_l^{n+1}}{\xi_l^n} = 1 - j\,\frac{c\,k}{h}\,\sin\left(k_l\,h\right)\ , \tag{7.45}$$

where $j = \sqrt{-1}$.

2-c) Deduce from Eq. (7.45) that this numerical scheme is *unconditionally unstable.*

3- The main purpose for using Eq. (7.44) was to obtain a numerical scheme of order 2 in space. Let's refrain our ardour and focus first on order 1 (in space) method by considering the following two numerical schemes

$$\frac{u_i^{n+1} - u_i^n}{k} \quad + \quad c\,\frac{u_i^n - u_{i-1}^n}{h} = 0 \tag{7.46}$$

$$\frac{u_i^{n+1} - u_i^n}{k} \quad + \quad c\,\frac{u_i^{n+1} - u_{i-1}^{n+1}}{h} = 0\ , \tag{7.47}$$

which are respectively explicit/implicit methods.

3-a) Insert a *Fourier* mode $\xi_l^n\, \mathrm{e}^{j\,k_l\,x}$ in the two above schemes to determine the corresponding amplification factors $\frac{\xi_i^{n+1}}{\xi_i^n}$ in both cases.

3-b) Show that for Eq. (7.46) and (7.47) we have respectively

$$\left|\frac{\xi_l^{n+1}}{\xi_l^n}\right|^2 \quad = \quad 1 - 2\,\alpha\,\left(1-\alpha\right)\,\left[1 - \cos\left(k_l\,h\right)\right] \quad \text{and} \tag{7.48}$$

$$\left|\frac{\xi_l^{n+1}}{\xi_l^n}\right|^2 \quad = \quad \left(1 + 2\,\alpha\,\left(1+\alpha\right)\,\left[1 - \cos\left(k_l\,h\right)\right]\right)^{-1}\ , \tag{7.49}$$

whith the *Courant* number defined as $\alpha = \frac{c\,k}{h}$.

3-c) Deduce from the previous question that:

1. Eq. (7.46) is stable if the CFL condition is fulfilled, i.e. $\alpha < 1$,
2. Eq. (7.47) is *unconditionally* stable.

3-d) Develop a computer program that solve the advection equation Eq. (7.37) together with the boundary conditions Eq. (7.38) based on one of the first order schemes Eq. (7.48) or (7.49). Chose the initial profile at your convenience. You could use

$$\overset{\circ}{v}(x) = \begin{cases} \sin^2\left(\frac{2\pi x}{L}\right) & \text{for } 0 \le x \le \frac{L}{2} \\ 0 & \text{for } \frac{L}{2} < x \le L \end{cases} , \tag{7.50}$$

for instance.

4- We modify slightly the scheme Eq. (7.44), which we recall is unstable, by replacing u_i^n by the mean $\dfrac{1}{2}\left(u_{i+1}^n + u_{i-1}^n\right)$ in the approximation of the temporal derivative. This new method, named after *Lax* and *Friedrichs*, is given by

$$u_i^{n+1} = \frac{1}{2}\left(u_{i+1}^n + u_{i-1}^n\right) - \frac{1}{2}\alpha\left(u_{i+1}^n - u_{i-1}^n\right) , \tag{7.51}$$

where α is the *Courant* number.

4-a) Determine the spatial and temporal orders of this method. Is it consistent ?

4-b) Study the stability of the scheme Eq. (7.51) in the *von Neumann* sense by inserting a mode $\xi_l^n\, \mathrm{e}^{k_l\, x}$.

4-c) Show that we have

$$\left|\frac{\xi_l^{n+1}}{\xi_l^n}\right| = \sqrt{1 - (1 - \alpha^2)\,\sin^2 k_l\, h} . \tag{7.52}$$

4-d) Deduce from Eq. (7.52) that the CFL condition, i.e. $\alpha < 1$, must again be fulfilled for the scheme to be stable.

4-e) Show that Eq. (7.51) can also be interpreted as an approximation to the following equation

$$\partial_t\, v(x,t) = -c\,\partial_x\, v(x,t) + \frac{1}{2}\frac{h^2}{k}\,\partial_x^2\, v(x,t) . \tag{7.53}$$

4-f) The diffusive term $\frac{1}{2}\frac{h^2}{k}\,\partial_x^2\,v(x,t)$ introduce *numerical diffusion* that stabilises the method. What is the price to pay for using artificial diffusion ? You can base you answer on the behaviour of a *Fourier* mode (see Eq. 7.52) as $n \to \infty$.

4-g) Write a computer program based on the *Lax-Friedrichs* method. Note that the scheme needs values on its left (u_{i-1}^n) but also on its right (u_{i+1}^n) which requires a careful implementation of the boundary conditions. You might want to use *ghost* cells on either side of the $[0, L]$ domain to ease this implementation.

7.4 Some useful hints concerning implementation

The idea of this section is to help you in developing the `C++` program needed in the projects of the previous sections. We take the diffusion equation as examples of 1D and 2D implementations. The extension to other type of equation is straightforward. In the very few situations where particular algorithm or tricks are needed for another type of equation we will introduce the necessary material.

We will try to make the programs as simple as possible in this section. If you are a `C++` expert feel free to express yourself. For "real" codes, perhaps the input/ouput part must be looked at with more attention, e.g. using a descent config file for input parameters is extremely usefull. For a large project, we also need to separate the code in several `.h` and `.cpp` files. Of course, for "real" codes, you might want to use the most advanced features of `C++`. With Objected Oriented programming the level of abstraction is high and generalisation of parts of the program are possible in a very elegant and efficient manner.

7.4.1 An appetizer : the 1D diffusion equation

The diffusion equation is our first numerical project and we are going to give you some hand for the implementation of our first numerical scheme given by Eq. (7.12).

First of all, we need to instruct the `autotools` that we add another program, say `heat1d`, to our project by modifying the `src/Makefile.am` file like this

```
bin_PROGRAMS=heat1d
heat1d_SOURCES=heat1d.cpp
```

where `heat1d.cpp` is the source of this program. Of course, if you already have some programs you just add `heat1d` to the list in `bin_PROGRAMS` by separating them by spaces.

Here is the simple structure of the `heat1d.cpp` file with all the necessary header files included and the use of the needed name spaces

```
#include <blitz/array.h>
#include <cmath>
#include <cstdlib>
#include <fstream>
#include <iostream>
#include <iomanip>

using namespace blitz;
using namespace std;

int main() {

  // We need to put some code in here !

  return EXIT_SUCCESS;
}
```

Now, in the core of the `main` function we will define the parameters of our program

```
  int nt=1000; // number of time steps
  int nx=100; // number of spatial steps
  double a=-5.; // left spatial boundary
  double b=5.; // right spatial boundary
  double t0=.1; // initial instant
  double t=t0; // time value
  double D=1.; // diffusion coefficient
```

Please, remember that in `C++` there is no need to declare these variables at the beginning of the code and you could define them wherever you want to. Here, we simply put them together for clarity.

We showed (see Eq. 7.14) that the stability of the FTCS scheme is expressed by $\frac{D\,k}{h^2} < \frac{1}{2}$. Consequently, we introduce the so-called *Courant* number $\alpha = \frac{D\,k}{h^2}$ and instead of giving the time step we set it according to the chosen value of α. We modify the code accordingly

```
  double alpha=.1; // Courant number
  double dx=(b-a)/(nx-1.); // Spatial grid size
  double dt=alpha*dx*dx/D; // time step
 // We issue a warning if the scheme is unstable
  if(alpha>=.5)
    cout << "WARNING: numerical scheme is unstable."
         << endl;
```

We need to store the values of the x_i's and u_i's as well as the values of the exact solution $v(x_i)$. For this purpose we will use `blitz++` arrays like this

```
// We declare them, 'nx' being their size
Array<double,1> x(nx),u(nx),v(nx);
```

Now, we must initialize these arrays, in particular we must start we `x` like this

```
for(int ix=0; ix<nx; ++ix) x(ix)=dx*ix+a;
```

Then, before we start we must code the initial condition to our problem which is as simple as this

```
 u=v=exp(-.25*x*x/D/t0);
```

We must loop on time steps to obtain our numerical solution. If we are not too careful we might do the following error

```
 for(int it=1; it<nt; ++it) {
   t+=dt;

   // FTCS scheme

   // ==========================================
   // THIS TWO COMMENTED LINES ARE WRONG !
   // ==========================================

   // for(int ix=1; ix<nx-1; ++ix)
   //   u(ix)+=alpha*(u(ix+1)+u(ix-1)-2.*u(ix));

   // Dirichlet boundary conditions

   double factor=sqrt(t0/t);
   u(0)=factor*exp(-.25*a*a/D/t);
   u(nx-1)=factor*exp(-.25*b*b/D/t);

   // Theoretical solution

   v=factor*exp(-.25*x*x/D/t);
 }
```

The loop between `ix=1` and `ix=nx-2` only is ok because we deal with Dirichlet boundary conditions so that we do not want to update `u(0)` and `u(nx-1)` according to the numerical scheme but specify them instead. So what is wrong and why ? As as an exercice take your time to identify the problem by yourself. For the impatient let's jump to the next paragraph.

Well indeed,

```
     u(ix)+=alpha*(u(ix+1)+u(ix-1)-2.*u(ix));
```

is wrong. The right hand side of this affectation does not use the values of u at the previous time step as it sould because in the `for` loop these values do change. To implement properly the numerical scheme we must store the values of u at the previous time steps. We need to declare an array before the `for` loop like this

```
Array<double,1> up(nx);
 for(int it=1; it<nt; ++it) {
   // code is omitted
}
```

and then replace the loop on `ix` by

```
    up=u
     for(int ix=1; ix<nx-1; ++ix)
       u(ix)+=alpha*(up(ix+1)+up(ix-1)-2.*up(ix));
```

to get the proper implementation of the numerical scheme.

For the readers knowing `C++` do not be tempted to declare and initialize the `up` array in a row like this

```
Array<double,1> up(u);
```

or equivalently like this

```
Array<double,1> up=u;
```

because in both cases we use the copy constructor. There is nothing wrong about this perhaps with other libraries but with `blitz++` we will break our code. The reason is that whenever we use the copy constructor with `blitz++`, the authors of the library decided that we actually copy the reference to the array, not the values, with the effect that it would be equivalent to the wrong code!

Basically that's all we need to do and actually we have produce our first working code except for one very important thing we have ignored so far: writing the numerical results to a file ! Obviously, it is quite useless to do some numerical computations if we do not store the results. We do this with the help of a file output stream that we declare like this

```
ofstream fout("heat1d.dat");
```

where `heat1d.dat` is the name of the output file.

Before entering the time loop, we must write the initial condition to the file like this

```
  fout << "# time=" << fixed
                    << setprecision(4)
                    << t << endl;
  for(int ix=0; ix<nx; ++ix)
    fout << x(ix) << " " << u(ix) << " " << v(ix) << endl;
fout << endl << endl;
```

where `<< fixed << setprecision(4)` is there because we want to store the time with 4 digits of precision. These are called stream manipulators and permits such things and much more. With `fout << endl << endl;` we add two blanck lines to separate the results between different time. This is needed because we are going to plot the results with a software called `gnuplot`.

Then, we must do the same thing for all other instant within the time loop with

```
int ntMovie=10;

// code is omitted ...

for(int it=1; it<nt; ++it) {

   // code is omitted ...

   if(!(it%ntMovie)) {
     fout << "# time=" << fixed
                       << setprecision(4)
                       << t << endl;
     for(int ix=0; ix<nx; ++ix)
       fout << x(ix) << " " << u(ix) << " " << v(ix) <<
           endl;
     fout << endl << endl;
   }
 }
```

where we have introduced a new `int` variable, `ntMovie`, that allows us to write the results only every `ntMovie` time steps. Well, yeah, it is called movie because we can produce a movie out of our results of course as we plan to do below.

To compile our code we simply use the `make` command and launch our program `heat1d` by invoking its name on the command line. At this point you should get a file named `heat1d.dat` containing our results.

7.4.2 Whose afraid of implicit schemes ?

For implicit schemes such as BTCS or *Crank-Nicolson* we must solve a linear system of equations for the u^{n+1}. In fact, for the 1D diffusion equation we obtain a tridiagonal system that we can solve by a direct method due to *Thomas*. In more general situation we cannot do that, especially for more than one dimension. There are some very efficient libraries to solve for linear system of equations such as

- `http://www.mcs.anl.gov/petsc/`,
- `http://www.netlib.org/lapack/`,
- `http://www.gnu.org/software/gsl/`,
- etc ...

and definitely we will not compete with them and advise you to use them in actual research work if you can. However, for this series of lectures we will use the *Gauss-Seidel* method because for finite difference equation it is quite easy to implement and it is furthermore sufficiently efficient for our present task. So let's go for it, even in 1D, though there exist better ways.

We define the *Courant* number by $\alpha = \frac{D\,k}{h^2}$. We can write the BTCS scheme Eq. (7.15) in the following way

$$u_i^{n+1} = \frac{1}{1+2\,\alpha}\left[\alpha\,\left(u_{i+1}^{n+1} + u_{i-1}^{n+1}\right) + u_i^n\right]\ . \tag{7.54}$$

In this particular case we do not have to write the linear system for the u_i^n explicitly. After a little algebra, you can convince yourself that Eq. (7.54) provide us the iterative algorithm showing up in the *Gauss-Seidel* method (see Sect. 3.3.1 for more details).

The algorithm can be summarized by

1. We initialize the solutions, i.e. $\bar{u}_i$, to u_i^n the value at the present time step.
2. Then we iterate on the $\bar{u}_i$ value according to the following relation obtained from Eq. (7.54)

$$\bar{u}_i = \frac{1}{1+2\,\alpha}\left[\alpha\,\left(\bar{u}_{i+1} + \bar{u}_{i-1}\right) + u_i^n\right]\ . \tag{7.55}$$

3. We stop when $\bar{u}_i$ as converged to a prescribed level of precision.

Yeah, it is simple but it has got a price. Since we have an iterative algorithm we introduce a supplementary error in the process. It is not that critical for the present application but it is better to be aware of this new source of error in case we need to control the level of precision more carefully.

Now in `C++` this algorithm can be traduced e.g. in the following way

```
  const int kmax=1000;
  double eps=1.e-8;
  double rerr=0.;
  double beta=1./(1.+2.*alpha);
  Array<double,1> ulast;

// ========================================
   // Gauss-seidel iterations
   // =========================

   int k=0;
   do {
        rerr=0.;
        ulast=u;

        for(int ix=1; ix<nx-1; ++ix) {
          u(ix)=beta*(alpha*(u(ix+1)+u(ix-1))+up(ix));
          double rdiff=fabs((u(ix)-ulast(ix))/ulast(ix));
          rerr=Max(rdiff,rerr);
        }
     } while(rerr>eps && ++k <= kmax);
```

We iterate until we reach the prescribed precision, here 10^{-8}, or we reach a given number of *Gauss-Seidel* iterations, just in case we were to greedy concerning the precision we can obtain.

We must define the `Max` function that does not exist in `C++` (before the `main` function) with

```
 template <typename T> T Max(const T& a, const T& b) {
     return (a>b ? a : b);}
```

Yes, we know. These are templates. Do not be scared though. This line just instructs the compiler to generate the `Max` and `Min` function for any type `T` (or class) that has comparison operators such as `<` or `>`.

We took the example of the BTCS sheme but of course, you can easily [2] adapt the above consideration and algorithm to the *Crank-Nicolson* scheme or any other scheme.

7.4.3 Going 2D

As an example, we will use the implementatoin of the 2D wave equation. Basically, going form 1D to 2D is as simple as using matrices instead of vectors.

First, let's code the basic structure of the code

```
#include <blitz/array.h>
#include <cmath>
#include <cstdlib>
#include <fstream>
#include <iostream>
#include <iomanip>

using namespace blitz;
using namespace std;
```

[2]Certainly we are afraid you have too if asked in class !

```
int main() {

  // Code missing ...

  return EXIT_SUCCESS;
}
```

We have all the necessary `includes` and `namespaces` set up. As in 1D we need to declare and initialize some variables

```
  int nt=1000; // number of time steps
  int ntMovie=10; // we write the results every 'ntMovie'
     time steps.
  int nx=128; // number of x steps
  int ny=128; // number of y steps
  double xMin=-5.; // min value for x
  double xMax=5.; // max value for x
  double yMin=-5.; // min value for y
  double yMax=5.; // max value for y
  double t0=0.; // initial time
  double c=1.; // propagation speed

  Array<double,1> x(nx),y(ny); // vector storing x and y
     values
  Array<double,2> u(nx,ny); // our numerical solution

  double dx=(xMax-xMin)/(nx-1.); // x step
  double dy=(yMax-yMin)/(ny-1.); // y step

  for(int ix=0; ix<nx; ++ix) x(ix)=dx*ix+xMin;  // x array
  for(int iy=0; iy<ny; ++iy) y(iy)=dy*iy+yMin; // y array

  double alpha=.1; //Courant number
  double h=dx*dy/sqrt(dx*dy+dy*dy);

  double dt=alpha*h/c; /time step set from alpha

  // Useful variables used below
  // that avoid to recompute these
  // quantities all the time.

  double alphaX=dt*c/dx;
  double alphaY=dt*c/dy;
  double alphaX2=alphaX*alphaX;
  double alphaY2=alphaY*alphaY;
  double alpha2=alpha*alpha;
```

Before we start to integrate, we need to handle the initial condition carefully. Remember that we want to implement a two-step method. In practice, this means we need to store not only the value of our numerical solution at t_n but also at t_{n-1} and t_{n-2}. We will use a Gaussian profile as an initial condition [3] like this

```
  Array<double,2> u0(nx,ny); // solution at t_{n-2}
  Array<double,2> u1(nx,ny); // solutino at t_{n-1}
  double sigma=1.; // sigma of the Gaussian
  double sigma2=sigma*sigma; // square of it

  // Gaussian profile

  double x0=0.; // x center of the Gaussian
  double y0=0.; // y center of the Gaussian

  // Gaussian initialisation

  for(int ix=0; ix<nx; ++ix) {
    double deltaX2=x(ix)-x0;
    deltaX2*=deltaX2;
    for(int iy=0; iy<ny; ++iy) {
      double deltaY2=y(iy)-y0;
      deltaY2*=deltaY2;
      u0(ix,iy)=.1*exp(-.5*(deltaX2+deltaY2)/sigma2);
    }
  }
```

We are not through yet because remember that the wave equation being order two in time we also need to initialize our numerical scheme by setting the derivative to given value, 0 in our case. This goes like this

```
  boundaryConditions(x,y,u0);
  // Derivative at t=0 is equal to 0
  for(int ix=0; ix<nx; ++ix)
    for(int iy=0; iy<ny; ++iy)
      u1(ix,iy)=(1.-alpha2)*u0(ix,iy)
        +.5*alphaX2*(u0(ix+1,iy)+u0(ix-1,iy))+
        + .5*alphaY2*(u0(ix,iy+1)+u0(ix,iy-1));

  boundaryConditions(x,y,u1);
```

We have introduce a helper function [4], `boundaryConditions` whose purpose is to implement the *Dirichlet* boundary condition. We impose that our solution must be 0 on the boundary. The declaration and definition of the function can be placed before the `main` function and looks like this

```
void boundaryConditions(Array<double,1>& x,
                        Array<double,1>& y,
```

[3]Note, that in this case we will not “respect” strictly our boundary conditions . . .

[4]In more fancy code, everything would be embedded in a class and we will use “methods” of this class.

```
                                    Array<double,2>& u) {

    int nx=x.size();
    int ny=y.size();

    for(int iy=0; iy<ny; ++iy) {
      u(0,iy)=0.; // left
      u(nx-1,iy)=0.; //right
    }

    for(int ix=0; ix<nx; ++ix) {
      u(ix,0)=0.; // bottom
      u(ix,ny-1)=0.; // up
    }

  }
```

Note the `&` symbol in the argument list of the function as in

```
  Array<double,1>& x
```

which means that we pass a reference on the array rather than the array value. We do this because for large arrays (i.e. large `nx` and `ny`), without the `&` we would ask to make a copy of the arrays and this can be very time consuming. In "pure" `C`, everythings is passed by reference so the way one does this is by passing the value of the pointer to the objects. However, we need to deference the pointer with the `*` operator or `->`. Sincerely, the `C++` reference is welcomed here because it simplifies everything and avoid the copy of large arrays.

We need to do the time integration with the following code

```
   double t=t0;

   double beta=2.*(1.-alpha2);
   for(int it=2; it<nt; ++it) {
      t+=dt;

      // Our numerical scheme

      for(int ix=1; ix<nx-1; ++ix)
        for(int iy=1; iy<ny-1; ++iy)
          u(ix,iy)=beta*u1(ix,iy)+
            +alphaX2*(u1(ix+1,iy)+u1(ix-1,iy))
            +alphaY2*(u1(ix,iy+1)+u1(ix,iy-1))
            -u0(ix,iy);

      // Dirichlet boundary conditions

      boundaryConditions(x,y,u);
      // We store the values at the previous time steps.

      u0=u1;
      u1=u;

      // Writing results every 'ntMovie' time steps.
      if(!(it%ntMovie)) writeResults(fout,x,y,u,t);
    }
```

We have introduce another function **`writeResults`** whose name is self-explanatory. We assume that an output stream as been defined like this

```
  ofstream fout("wave2d.dat");
```

in order to write our results in a file. The definition of the function is the following

```
void writeResults(ofstream& fout,
                  Array<double,1>& x,
                  Array<double,1>& y,
                  Array<double,2>& u,
                  double t
                  ) {

  int nx=x.size();
  int ny=y.size();

   fout << "# time=" << fixed << setprecision (4) << t <<
      endl;

   // Data gnuplot format

   for(int ix=0; ix<nx; ++ix) {
     for(int iy=0; iy<ny; ++iy)
       fout << x(ix) << " "
            << y(iy) << " "
            << u(ix,iy) << endl;
     fout << endl;
}
   fout << endl;
}
```

Of course, you can write your results as you like. In the present case we are trying to please our **`gnuplot`** friend. If we want to have the solution at the two first time steps we must write them to the file before the time loop like this

```
  writeResults(fout,x,y,u0,t);
  t+=dt;
  writeResults(fout,x,y,u1,t);
```

right after the `double t=t0;` declaration.

That's it. If you want to have a look at your results, and even produce a movie out of your data you need to have a look at the next section about `gnuplot`.

7.5 Visualizing the results with gnuplot

7.5.1 Simple 2D plots

To visualize our results now we will use the `gnuplot` software. We already prepare our files to please `gnuplot` so it is actually quite easy to produce a plot for a given time step. First you need to start `gnuplot` by invoking its name on the command line and then at the `gnuplot>` prompt type this

```
plot "heat1d.dat" using 1:2 index 20
replot "heat1d.dat" using 1:3 index 20 with lines linecolor
    3
```

which instruct `gnuplot` to plot our numerical approximation as red crosses and the theoretical solution superimposed in a continuous blue line.

At the time of writing these lecture notes, both authors were not `gnuplot` users at all and we chose it for its famous simplicity of use. At first site, you might find the look of the default figures produced with `gnuplot` quite uncertain to put it that way. Yeah, that's true but with some bit of effotr you can obtain some terrific plots that you will not be ashamed of showing in a publication. Everything is a matter of taste of course, but let's try this

```
plot "heat1d.dat" using 1:2 i 20 lw 2
replot "heat1d.dat" using 1:3 i 20 w l lc 3 lw 3 lt 1
set tics out
set tics scale 1.5
set mxtics 5
set mytics 5
set pointsize 1.5
set border linewidth 1.5
unset key
set size square
set xlabel "x"
set ylabel "v(x)"
set title "Heat equation"
replot
```

You might want to put the results in a postscript file for instance, in that case you could try this

```
set terminal postscript eps enhanced color font "Bookman-
    Light" 18
set output "heat1d.eps"
replot
```

and you can have a look at the results if you have the `gv` command installed (most certainly on a `linux` system you have) with

```
gnuplot> !gv heat1d.eps
```

We can do much better concerning the quality of the plot. With gnuplot we can generate a LaTeX file that once compiled produce a better postscript file like this

```
set xlabel "$x$"
set ylabel "$v(x)$"
set terminal epslatex standalone color
set output "heat1d.tex"
replot
```

Note that for e.g. the x label we used `$x$` which is the way to handle mathematical formulas in LaTeX. In our case it has only the effect of selecting the special fonts used my LaTeX to print formulas. However it would be possible to put any kind of mathematical symbols or formula as we would do with LaTeX except that we must escape the "\" symbol. For instance to print the fraction $\frac{x}{y}$ you need to use `$\\frac{x}{y}$`.

Then, we need to flush [5] the `.tex` output file with

```
unset output
```

or simply exit `gnuplot` and do the following on the command line

```
> latex heat1d.tex
> dvips -E heat1d.dvi -o heat1d.eps
```

to produce an encapsulated postcript file that is shown in Fig. (7.1).

7.5.2 3D plots

"3D" plot means that we plot a function of the *two* variables x and y for instance. We can generate some color maps or plot the surface of the function or a mix of both. Again, with `gnuplot` it is relatively easy. Let's use our results for the 2D wave equation.

If you just want to have a look at the data, say for the time step 200, it is indeed quite easy. You just need to invoke the two following command

[5] This is a trick without which the LaTeX file would not be complete and could not be compiled.

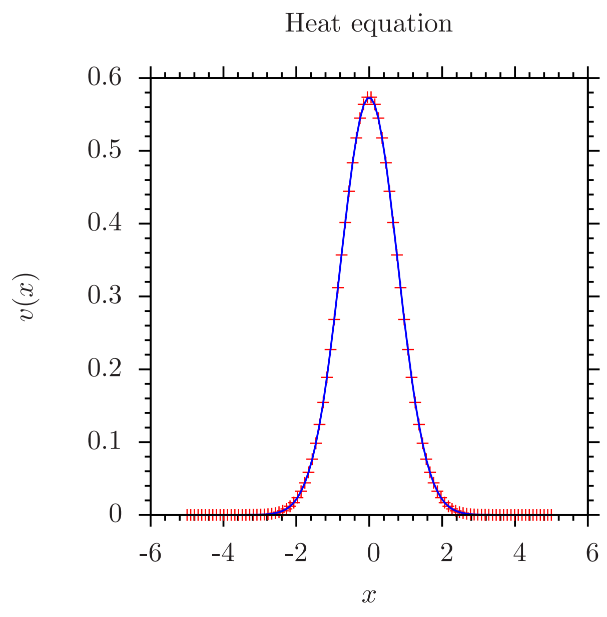

Figure 7.1: `eps` figure produced with `gnuplot`.

```
set pm3d map
splot "wave2d.dat" index 200
```

`pm3d` is a plotting style for the visualisation of 3D data. Here we require to plot a map and we tell `gnuplot` to do so with the first line. Then, the real plotting is carried out with `splot`.

If you feel that this is very easy perhaps you also think it is not very pretty. However, you can again use the `epslatex` terminal to get “quality-of-publication” `eps` figure, I think. You can do this with

```
set tics scale 1.5
set pointsize 1.5
set border linewidth 1.5
set tics out
set mxtics 5
set mytics 5
set mcbtics 5
set nokey
set size square

# Needed to plot the contours

set cntrparam cubicspline
set cntrparam level auto 10
unset clabel
```

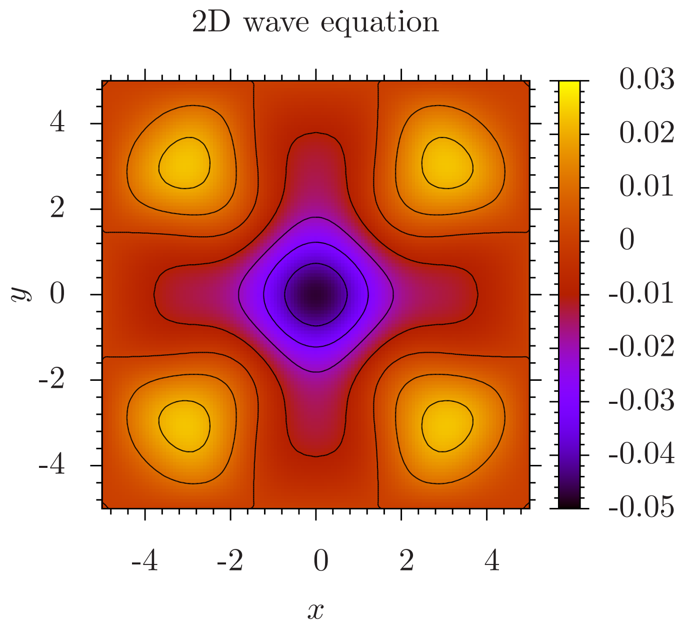

Figure 7.2: 2D wave equation. This image is obtained with the `epslatex` terminal.

```
set contour base
set title "2D wave equation"

set xrange [-5:5]
set yrange [-5:5]
set xlabel "$x$"
set ylabel "$y$"
set pm3d map

splot "wave2d.dat" index 200 linewidth 2 lc rgbcolor "black
    "

set terminal epslatex standalone color
set output "wave2d.tex"
replot
unset output

!latex wave2d.tex
!dvips -E wave2d.dvi -o wave2d.eps
```

You can see the results in Fig. (7.2).

For some results, the map representation of Fig. (7.2) is probably not the best one. Since the 2D wave equation with fixed *Dirichlet* boundary conditions is a model for the movement of a square drum skin we might want to represent the surface of the skin. We can do that in `gnuplot` with the `pm3d` style. We must change the plotting instructions in the previous `gnuplot` script in that way

```
set pm3d
set xyplane at 0.
unset contour
splot "wave2d.dat" index 210 every 3:3 with lines lc
    rgbcolor "black"
```

We have added an option to the `splot` command : `every 3:3`. We have a spatial grid of 100×100 which is too much if we want to overlay the surface mesh because we would fill the whole surface with the lines that trace the surface mesh. With `every 3:3` we tell `gnuplot` to plot only every 3 data blocks (corresponding to a given x) and every 3 lines (corresponding to a given y) within each data block. This is coherent with the way we wrote the data, the so-called "grid data" format of `gnuplot`. Ok, we do not show all the available information but at least we get a reasonably neat and clean plot.

It is just an example of what you can do with `gnuplot` and finally data representation is a subject by itself clearly out of the scope of this course. One has to play and see what is the best representation for the data under consideration. I found that pictures as in Fig. (7.3) are best suited when producing animation because the surface mesh help in seeing the deformations of the "drum skin". Finally, it is a matter of taste and you will have to experiment with `gnuplot` and find your way.

7.5.3 How to generate a movie

Since our results are time dependent the natural way of visualizing them is to generate a movie. We chose to use a mixed of `bash` scripting and `gnuplot` to produce all the necessary pictures (in `.png` format). For the encoding of the movie we use the `avcon` library, previously known as `ffmpeg` that you can found here `http://libav.org` so you must check that it is installed on your system [6].

Here is the `bash` script for generating a movie out of our output file. It is kind of cryptic, certainly not on purpose, but it is unfortunately unavoidable with shell scripting.

Take it as a template that you can adapt to suit your own preferences and needs. We will definitely not explain how it works because `bash` scripting is

[6] If it is old enough, it is certainly still called `ffmpeg`.

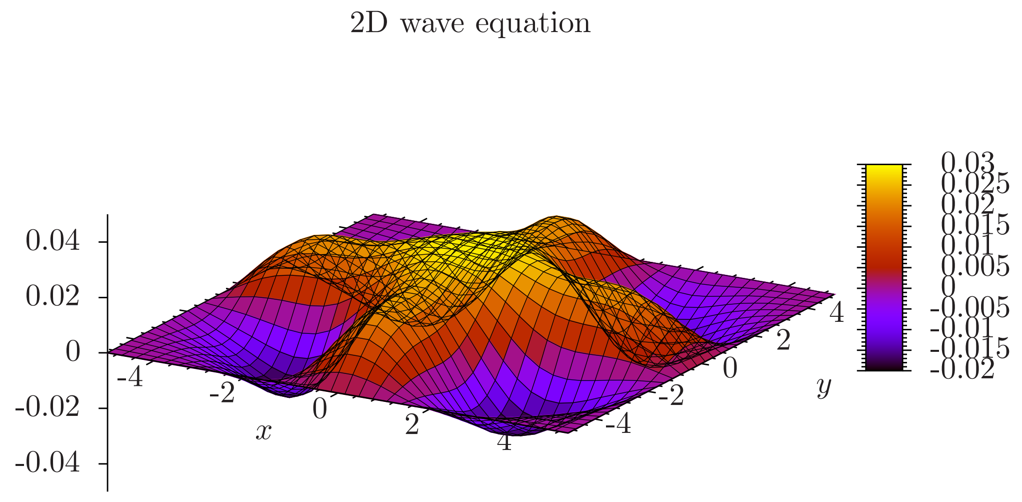

Figure 7.3: 2D wave equation with surface plot and mesh.

clearly out of the scope of these lecture notes. We apologize for the "esoteric" nature of the script but here it goes

```bash
#!/bin/bash

declare -a timeArray
timeArray=(`grep 'time=' $1 | sed 's/ //g' -`)
nTime=${#timeArray[*]}

for ((i=0; i<$nTime; ++i)); do
    timeArray[$i]=${timeArray[$i]##*=}
done

N=$(grep '^$' $1 | wc -l)
((N/=2))

nDigits=0;
for ((i=N; i>=1; i=i/10)); do
    ((++nDigits));
done
((--nDigits))

for ((i=0; i<$N; ++i)); do
```

```
    file=${1%.*}`printf "%0${nDigits}d" "$i"`".png"
    echo "Creating "$file
    gnuplot << GNUPLOTSCRIPT

set terminal png enhanced transparent font 'Times,18'
set output "$file"

set style line 1 lc rgbcolor "blue" lt 1 lw 1
set style line 2 lc rgbcolor "red" lt 1 lw 2
set tics out

set tics scale 2.
set mxtics 5
set mytics 5
set pointsize 2.
set border linewidth 2.

set key box reverse

set yrange [0.:1.]
set xlabel "x"
set ylabel "v(x)"
set title "Heat equation (t=${timeArray[$i]})"

plot  '$1' using 1:2 index $i linestyle 1 title "FTCS", \
'$1' using 1:3 index $i ls 2 with lines title "exact"

GNUPLOTSCRIPT
done

output=${1%.*}"%"0${nDigits}d".png"
movie=${1%.*}.avi

printf "Encoding the '${movie}' file ..."

avconv -y -f image2 -i $output -b 8192k $movie &>/dev/null

printf " done !\n"
\rm ${1%.dat}*.png
```

Let's name this script `makemovie` for instance. Do not forget to make it executable with

```
> chmod +x makemovie
```

and use it like this to produce a `heat1d.avi` file

```
> makemovie heat1d.dat
```

that you can visualize with your favourite movie player (e.g. for `linux`: `xine`, `totem`, `vlc`, ...).

We can now generate a movie for 2D data. However, this `bash` script must be adapted when dealing from one kind of data to another, e.g. for 3D data: map or surface plots. Of, course with `bash` you can generalised everything as you like and develop a script that will handle any kind of situations and data. We will not enter into all this details but the above script can be the basis for fancier `bash` script of your owrn.

Bibliography

Abramowitz, M. & Stegun, I. 1964 (Dover publications)

Bohren, C. F. & Huffman, D. R. 1998, Absorption and Scattering of Light by Small Particles, ed. Bohren, C. F. & Huffman, D. R.

Eckel, B., Allison, C., & Allison, C. 2000, Thinking in C++, vol. 2 (Pearson Education)

Eckel, B., Allison, C., & Allison, C. 2003, Thinking in C++, vol. 2 (Pearson Education)

Goldberg, D. 1991, ACM Computing Surveys, 23, 5

Gregoire, M., Solter, N. A., & Kleper, S. J. 2011, Professional C++ (John Wiley & Sons)

Henrici, P. 1962, Discrete variable methods in ordinary differential equations, ed. Henrici, P.

Knuth, D. 1981, The Art of Computer Programming (Volume 2) (Addison–Wesley)

LeVeque, R. 2007, Finite difference methods for differential equations (Society for Industrial and Applied Mathematics Philadelphia)

Press, W., Flannery, B., Teukolsky, S., Vetterling, W., et al. 1986, Numerical recipes, Vol. 547 (Cambridge Univ Press)

Ralston, A. & Rabinowitz, P. 2001, A First Course in Numerical Analysis (Dover Publications, Inc.)

Trefethen, L. 1996, Finite difference and spectral methods for ordinary and partial differential equations (unpublished text)