Numerical Methods

Blat, Gimenez, Perea, Giraldo

Universitat Pompeu Fabra, 2021

Contents

Chapter 1

Introduction and Basic Concepts

1.1 Basic Concepts

Numerical Methods (NM) is a part of Mathematics mostly devoted to obtain (approximate) solutions of equations in a practical way. We recall that an equation usually involves one or more unknowns in an equality, and that the value(s) of the unknown(s) that satisfy the equation are called solutions. With NM one usually obtains numerical values which approximate the exact solution of the equation.

One of the concerns of NM is to obtain solutions *efficiently*, i.e., by using algorithms (procedures to compute the solution) which are as fast as possible.

Another important concern is to estimate the *errors*, i.e., how much the *numerical* solution differs from the *exact* solution of the equation. A source of errors which always exists is due to the fact that most real numbers can only be represented in an approximate way in the computer. This type of errors are called *rounding* errors. Other errors can come from the computer operations themselves, which are sometimes based on approximate algorithms. The algorithms can be another source of errors. These errors derived from the algorithms are called *discretization* errors. We shall study the so called *convergence* and *stability* of the algorithms as well.

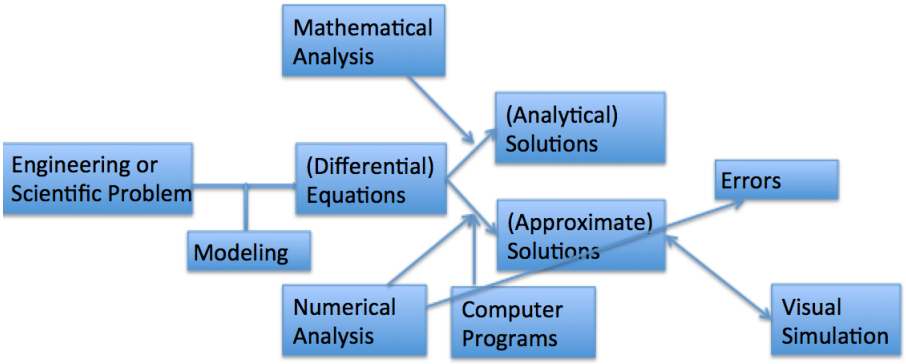
A final important aspect is the quality of the computer code. Fundamental properties are *reliability* and *robustness*; good programs should be *portable* and *maintainable* as well.

1.2 Orientation of the Course

The focus of this course is on numerical methods for *Differential Equations* (DEs), as applications of DE are important and provide a stimulating motivation. The most interesting DEs from the point of view of the applications we deal with stem from modelling physical phenomena. An important current application of this type of DEs is **visual simulation**. Numerical solutions of DEs are frequently used to generate the visual effects appearing in films and games. Some examples that we will consider in this course are the simulation of water and smoke. From another point of view, the visual simulation should also help us to understand better the errors, as well as the convergence and stability concepts.

The consideration of efficiency that we mentioned is especially important, for instance, in games, where the simulation has to be real-time, i.e., it has to produce around 30 frames per second, which means that a result has to be obtained in 1/30th of a second.

In order to understand better the applications, we shall discuss the physical models and the derivation of the corresponding DEs, and, if convenient, their *exact* solutions, also called *analytical* solutions, which we distinguish from their *approximate* or *numerical* solutions. If the DEs include partial derivatives they are called *Partial Differential Equations* (PDEs); if not, they are called *Ordinary Differential Equations* (ODEs). The following image represents the different fields involved in this process.



1.3 Basic ODE Concepts

DEFINITION 1.1
Let x be a function of one variable, usually denoted as t, which is n times differentiable. An expression such as $F(t, x, x', \ldots, x^{(n)}) = 0$, $n \geq 1$, is called an ***(ordinary) differential equation*** *and n is the* ***order*** *of the equation.* □

An example of first order ODE obtained from a physical model is the DE that represent the uniform movement of a particle: $x'(t) = v$, where v is constant. The ODE representing the uniformly accelerated movement, $x''(t) = a$ with constant a, is second order.

A function $f(t)$ such that $F(t, f(t), f'(t), \ldots, f^{(n)}(t)) = 0$ for every t is a **solution** of the equation. We call this solution *exact* and sometimes *analytical*. **Remark** that the solution of a DE is a *function*, when the solution of an algebraic equation is usually just a *number* (or several of them, one for each unknown).

The *numerical/approximate solution* calculated with a computer through a numerical method cannot be obtained for all the infinite values of t: we can only obtain the solution for a finite number of values of t, usually $t_n = n\Delta t$ with Δt fixed; and the solution is only an approximation of the exact solution.

In the expressions used so far, we are implicitly assuming that a solution of the ODE exists and, furthermore, that it is unique. Indeed, this is the case for ODEs coming from physical models and under suitable conditions. Mathematically, Picard's theorem shows that, under some specific hypotheses on the function f, the solution of a first order ODE $\frac{dx}{dt} = f(t, x)$ with a given **initial value** $x_0 = x(0)$ exists and is unique within a certain interval of time, i.e., for $t \in (0, a)$. A successive approximation technique is generally used to prove this theorem.

When the initial value is not fixed, the so called **general solutions** contain one or more arbitrary constants.

EXERCISE 1
Couple each ODE with its general solution in the following table, using the definition of solution. Remark the relationship between order of the DE and number of arbitrary constants.

1	$x' = 0$	A	$x = t^2 + t + C$
2	$x' = 2t + 1$	B	$x = 2t^2 + C_1 t + C_2$
3	$x' = x$	C	$x = -5 + Ce^{-2t}$
4	$x' + 2tx = t$	D	$x = C_1 sin(2t) + C_2 cos(2t)$
5	$x' + 2x + 10 = 0$	E	$x = C$
6	$x'' + 4x = 0$	F	$x = \frac{1}{2} + Ce^{-t^2}$
7	$x'' = 4$	G	$x = Ce^t$

1.4 Analytical Solutions through Separation of Variables

If a first order DE can be written in the form $f(x)dx = g(t)dt$, i.e., the variables x and t are separated, the analytical general solution of the equation is $\int f(x)dx = \int g(t)dt + C$, as can be easily checked by derivation. The **separation of variables** consists in rewriting a given DE in the form of separated variables.

Exercise 2
Find the general solution of the ODE $x' = x$ using separation of variables.

1.5 Models and Corresponding ODEs

We do not recall here the basic cinematic differential equations, a pre-requisite for Physics, but we assume them well known to the students.

1.5.1 Newton's Cooling Law

Newton formulated the following *Law of Cooling*: The rate of loss of heat of a body is proportional to the temperature difference between that of the body and that of its surroundings, provided the difference is small.

This law is represented through a DE: t is the time; $T(t)$ the body temperature at t; the cooling, represented by $C(t)$, is the rate of change of the body temperature; the initial body temperature is T_0; the surroundings temperature is T_S. The ODE describing this model is:

$$C(t) = \frac{dT}{dt} = c(T_S - T(t))$$

where c is the cooling constant. Through separating the variables T and t and integrating, one obtains:

$$T(t) = T_S - (T_S - T_0)e^{-ct}$$

Reflection: What happens if $T_S > T_0$?

Stationary solutions

Let us impose $\frac{dT}{dt} = 0$. Then $c(T_S - T) = 0$ and as $c \neq 0$, $T_S - T = 0$ and this equation has the solution $T = T_S$. If a DE has a solution which does not depend on time, it is called **a stationary solution** (of the DE). This type of solutions is obtained through making the velocity, the change of temperature, 0, as just seen. Thus, if the initial temperature is T_S, it will not change, it will remain stationary. *Note* that, under this physical model, any initial temperatute tends to reach the stationary one after time.

Exercise 3
Using Newton's cooling law, $T'(t) = k(T_S - T(t))$:

- *The temperature of some food before introducing it in a freezer, which is at -10^o, is 25^o. After 8 minutes the food is at 0^o, when will it reach -5^o?*
- *Which is the stationary solution of the equation?*

1.6 Euler's Method for First Order ODEs

1.6.1 Introduction: a simple example

The numerical solution of ODEs is also known as *numerical integration* of ODEs. A basic method of numerically solving DEs is the *finite diferences method*. Euler's method exemplifies finite differences.

Let us start with a simple case, where the ODE is:

$$\frac{dx}{dt} = x'(t)$$

and the initial value is $x(0) = x_0$. By the definition of derivative at a point:

$$x'(t) = \lim_{\Delta t \to 0} \frac{x(t + \Delta t) - x(t)}{\Delta t}$$

If we take a small Δt, one could write, approximately:

$$x'(t) \simeq \frac{x(t + \Delta t) - x(t)}{\Delta t}$$

If we replace $\simeq$ by $=$ (thus, an approximation is made, and an error is introduced):

$$x'(t)\Delta t = x(t+\Delta t) - x(t)$$

$$x(t+\Delta t) = x(t) + \Delta t x'(t)$$

The right hand side (RHS) of the previous equality are the first two terms of Taylor's expansion of the left hand side (LHS). With a fixed small Δt, the formula can be used as a method to compute (approximate) values of the solution at instants $n\Delta t$, through iterations:

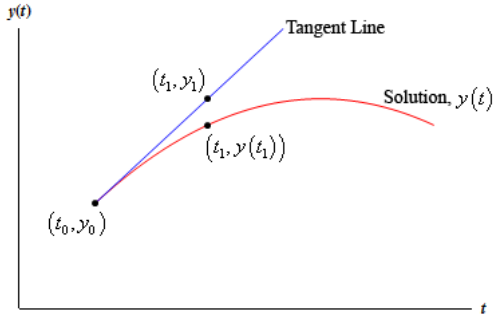
- start with the initial value $x(0) = x_0$
- compute the value at Δt

  $$x(\Delta t) = x(0) + \Delta t x'(0)$$

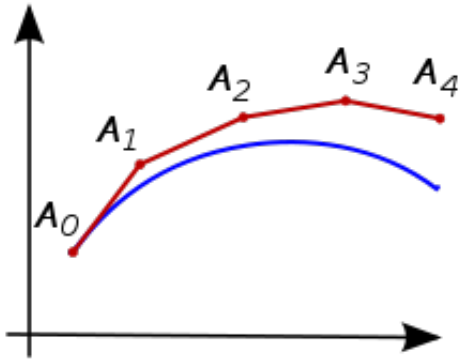
- and iteratively, compute the value at $n\Delta t$ using the value at $(n-1)\Delta t$ (each of these iterations is called a **step**):

  $$x(n\Delta t) = x((n-1)\Delta t + \Delta t) = x((n-1)\Delta t) + \Delta t x'((n-1)\Delta t)$$

The following image (taken from http://tutorial.math.lamar.edu/) visualises the computation of the first step, showing both the approximate and the exact solutions, where the unknown is called y instead of x.



The following image (taken from the Wikipedia) sketches the computation of four steps, showing the approximate solution in *red* - interpolating linearly between the four computed values -, and the exact one in *blue*.



Remark that

- the exact and approximate solutions are only equal at the initial time,
- the approximation used to compute the next value is based on the tangent
- the *error*, i.e., the difference between the exact and the approximate solution, is increasing with every step.

EXERCISE 4
Consider the ODE $x'(t) = x(t)$ with initial value $x(0) = 1$.

- *Check that e^t is the solution of this IVP.*
- *Compare the values obtained through Euler's method at times $t = 0.1, 0.2, ..., 1$ (i.e., using $\Delta t = 0.1$) to the exact values (e^t) at those times.*
- *Compute the absolute and relative errors in the interval $t \in [0, 1]$.*

1.6.2 Formalisation of Euler's Method in the more general case

Consider a slightly more general first order ODE, where the derivative can be isolated, and an initial condition:

$$\frac{dx}{dt} = x'(t) = f(t,x) \qquad x_0 = x(0)$$

To express the numerical method more clearly, it is convenient to improve the notation. Let us denote $\Delta t = h$, $kh = t_k$, and x_k the approximate solution at t_k (the exact solution at t_k is $x(t_k)$). Then, the approximation that we wrote above in the simpler case, $x(n\Delta t) = x((n-1)\Delta t) + \Delta t x'((n-1)\Delta t)$, can be written for this case with the new notation as:

$$x_{k+1} = x_k + hf(t_k, x_k) \qquad x_0 = x(0)$$

This is an *iterative* formulation, where the computation of the approximation in the (next) step of time requires the approximation computed in the previous step. This is the (forward) *Euler's method.*

1.6.3 Basic Error Concepts

The source of error of Euler's method is that the exact derivative, $f(t, x(t))$, is replaced by an approximation, the fraction $\frac{x(t+h)-x(t)}{h}$. Then the *local discretization error* (of Euler's method) at each t is defined as the difference between the approximation and the exact derivative, i.e.:

$$L(t,h) = \frac{x(t+h) - x(t)}{h} - f(t, x(t))$$

Indeed, if we consider only one iteration, the *error* made, the difference between the exact and the approximate solutions, would be:

$$x(t_{k+1}) - x_{k+1} = x(t_{k+1}) - x(t_k) - hf(t_k, x(t_k)) = hL(t_k, h)$$

In practice, $x(t_k))$ is not known, what has been previously computed is its approximation x_k; the *global discretization error* is the error accumulated in n iterations. In later chapters, errors will be studied in more detail, and more precisely: for instance, showing that under some conditions, the global discretization error of Euler's method is $0(h)$, i.e., tends to 0 when h tends to zero.

Exercise 5
Consider the ODE $x'(t) = x^2(t) + 2t - t^4$ with initial value $x(0) = 0$.

- *Check that t^2 is the solution of this IVP.*
- *Compare the values obtained through Euler's method at times $t = 0.1, 0.2, ..., 1$ (i.e., using $\Delta t = h = 0.1$) to the exact values at those times.*
- *Determine the errors in the interval $t \in [0, 1]$.*

Exercise 6
Find the general solution of the ODE $x' - t = sin(t)$ using separation of variables.

Exercise 7
The dynamics of an infectious bacterial population $P(t)$ within a body evolves according to the ODE:

$$P'(t) = (kcos(t))P(t), k > 0$$

- *Obtain a general solution of the population evolution.*
- *If $k = 1$ and the initial population is 100, provide the dynamics of the population.*
- *Sketch the graphical representation of this solution.*

Exercise 8
Attempt to find the general solutions of some of the ODEs in exercise 1 through separation of variables.

1.7 Concepts for Programming: State, Simulation, Visualization

A physical model is represented through DEs involving several variables: those representative of the model are called in programming the **state representation**. Through numerical methods, the DEs can be approximately solved, i.e., we say then that the system is being **simulated**. The simulation can be visualised: we speak of the **state visualisation**. In this course, we will be using mostly Python, as well as MatLab at the end of the term, and we might mention the programming environment *Processing*, because it makes the visualization especially easy. State visualisation is discussed in the next chapter, while simulation in the following ones.

1.8 Supplementary Material

1.8.1 Population Dynamics

Malthus' model

The *Malthus' model* postulates that the population grows proportionally to its current size. If $u(t)$ is the population, Malthus' model can be represented through the ODE:

$$\frac{du}{dt} = ku$$

Separating variables one gets:

$$\frac{du}{u} = kdt$$

and thus

$$\log u = kt + C \qquad u = e^{kt+C}$$

If the initial population is $u(0) = u_0$ we obtain

$$u = u_0 e^{kt}$$

Which is the expression of the solution. Thus, Malthus' model results into an *exponential growth* of the population.

Logistic growth

The *logistic growth model* (or Verhulst's model) is represented by the ODE:

$$\frac{du}{dt} = ku(1-u)$$

and initial population $u(0) = u_0$. Separating variables:

$$\frac{du}{u(1-u)} = kdt$$

which can be integrated descomposing the fraction as:

$$\frac{1}{u(1-u)} = \frac{A}{u} + \frac{B}{1-u}$$

Identifying coefficients, one obtains $A = B = 1$, which leads to:

$$\log \frac{u}{1-u} = kt + C \qquad \frac{u}{1-u} = e^{kt+C}$$

and thus:

$$u = e^{kt+C}(1-u) \qquad u = \frac{e^{kt+C}}{1+e^{kt+C}}$$

Visualising u versus t, a characteristic sigmoidal curve appears, the logistic curve of growth, instead of the exponential growth of Malthus' model.

If we impose $\frac{du}{dt} = 0$ (velocity of change 0), then $u(1-u) = 0$ and this equation has two solutions, $u = 0$ and $u = 1$, which are the *stationary solutions* of the DE. Thus, if the initial population is 0 or 1, the population will not change, it will remain stationary.

Any nonzero initial population tends to reach the population limit $u = 1$. Indeed, if the initial population is larger than 1, it decreases; if it is smaller, it grows.

Predator-prey models

The *Lotka-Volterra model* considers two populations: the *prey* $u(t)$ and the *predator* $v(t)$; the prey grows proportionally to its size, but decreases proportionally to the size of the predator and its own size; the predator decreases proportionally to its size, and grows proportionally to the size of the prey and its own size. This can be represented through the system of DEs:

$$\frac{du}{dt} = \alpha u + \beta uv$$

$$\frac{dv}{dt} = \gamma v + \delta uv$$

where $\alpha > 0$, $\beta < 0$, $\gamma < 0$ and $\delta > 0$ (positive sign means growth, negative sign decay).

1.8.2 Trajectory in terms of speed

The movement of a particle along a line, the X axis for instance, can be represented through a first order ODE, where $v(t)$ is the velocity or speed at each instant t:

$$\frac{dx(t)}{dt} = v(t)$$

The analytical/general solution of this DE can be obtained through variables separation. The DE can be rewritten as:

$$dx = v(t)dt$$

and integrating both sides of the equality:

$$x(t) = \int v(t)dt + C$$

If $v(t)$ were constant, $x(t) = vt + C$; depending on the complexity of the function $v(t)$, a closed solution (i.e., a solution given by a mathematical expression) might not exist. It seems that there are infinitely many solutions of the DE, one for each value taken by C. But considering the physical model, the solution is in fact unique, as the particle is initially (at $t = 0$) at a given position, namely, $x(0) = x_0$, and thus $C = x_0$. The unique solution, considering the initial position, is:

$$x(t) = \int_0^t v(t)dt + x_0$$

and $x(t) = \int v(t)dt + C$ is the general solution.

When both the ODE and the initial condition are given, we say that we have a **Cauchy problem** or an **initial value problem (IVP)**.

1.8.3 Trajectory in terms of acceleration

The physical model of the movement is more meaningful in terms of the acceleration $a(t)$: the (second order) ODE representing the movement is:

$$\frac{d^2x}{dt^2} = a(t)$$

Second order ODEs are usually solved through two steps, each first order. Namely, if we call $\frac{dx(t)}{dt} = v(t)$ then the second order ODE can be represented as a first order ODE in v:

$$\frac{dv(t)}{dt} = a(t)$$

Solving this ODE through separation of variables:

$$v(t) = \int_0^t a(t)dt + v_0$$

and reintroducing $\frac{dx(t)}{dt} = v(t)$, one has:

$$\frac{dx(t)}{dt} = v(t) = \int_0^t a(t)dt + v_0$$

which results through separation of variables in:

$$x(t) = \int_0^t \int_0^t a(t)dt + v_0 t + x_0$$

If $a(t)$ were constant, the known expression for the uniformly accelerated movement results:

$$x(t) = \frac{1}{2}at^2 + v_0 t + x_0$$

The same equations and solutions hold for the more realistic situation of three-dimensional movement, considering $x(t)$, $v(t)$ and $a(t)$ as 3D points or 3D vectors, as appropriate.

1.8.4 The Trajectory of a Rocket

A rocket can fly long distances due to the consumption of its own fuel. This means that its mass varies during the flight. A relatively simple model of the trajectory of a rocket takes into account only three forces: a vertical one due to *gravity*; the *thrust* of the rocket T assumed to be constant while the fuel lasts; and *friction* of the atmosphere, proportional to the rocket speed v and its section s but of negative sign. This model does not consider the thrift due to the wind, for instance. Other simplifications are to assume that the trajectory is not too long nor too high, so that the Earth can be considered as plane, the air density and the gravity as constant.

Then, the trajectory is planar, so that the rocket position can be represented by means of just two coordinates, x horizontal and y vertical; similarly, velocity is represented through v_x and v_y, acceleration through a_x and a_y ; θ is the angle between $\vec{v}(t)$ and a horizontal line; and ρ is the air density. Using the law of classical mechanics

$$\frac{d(m\vec{v})}{dt} = \vec{F}$$

the following ODEs are obtained:

$$\ddot{x} = \frac{1}{m}(T - \frac{1}{2}c\rho s v^2)cos\theta - \frac{\dot{m}}{m}\dot{x}$$

$$\ddot{y} = \frac{1}{m}(T - \frac{1}{2}c\rho s v^2)sin\theta - \frac{\dot{m}}{m}\dot{y} - g$$

where: $\dot{x} = \frac{dx}{dt}$, $\ddot{x} = \frac{d^2x}{dt^2}$, $\dot{y} = \frac{dy}{dt}$, $\ddot{y} = \frac{d^2y}{dt^2}$, $\dot{m} = \frac{dm}{dt}$, $\ddot{m} = \frac{d^2m}{dt^2}$. Indeed it is a system of coupled second order DEs.

Chapter 2

State, Simulation, Visualization, *Processing*

A physical model is represented through DEs involving several variables: those representative of the model are called the **state representation**. Through numerical methods, the DEs can be approximately solved, i.e., the system can be **simulated**. The simulation can be visualised: in this chapter we introduce the **state visualisation**. We shall use the programming environment *Processing* in this course, because it makes the visualization especially easy. Simulation is discussed in depth in other chapters.

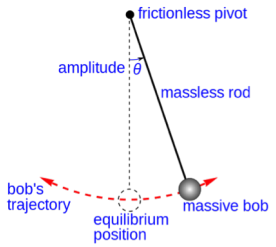
2.1 State

2.1.1 Representing the State

The first step towards the visualization of a (physical) system is to decide the **state** description, i.e., the variables that represent the system at each time, in a way which is both *complete*, i.e., without missing anything needed; and *economical*, i.e., without unnecessary data, which would need extra computation and storage.

2.1.2 A First Example: the Pendulum

Assume we wish to simulate the pendulum movement restricted to two dimensions. Two variables are sufficient to represent the state, namely, θ, the angle of the pendulum with respect to the equilibrium at each time; and the angular speed θ'. An extra fixed magnitude is needed for the representation: the pendulum arm length. The following image, taken from Wikipedia, shows a multiplicity of variables related to the pendulum, while some of them, those just mentioned, are enough to represent the state.



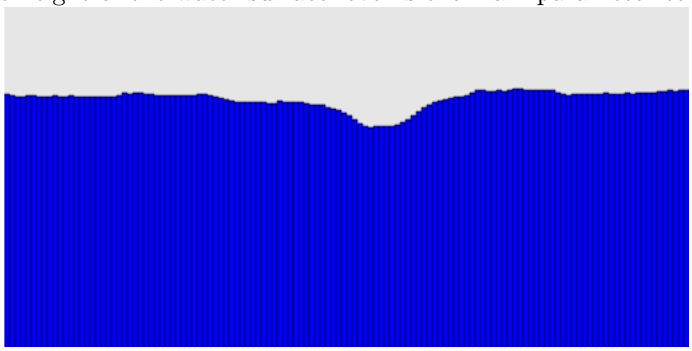
The following Processing code can represent the pendulum state:

```
# Length of the pendulum arm, in metres. In fact, it is a constant
# which does not belong really to the simulation state
PendulumLength = None

# Angle of the pendulum at each instant, in radians
StatePendulumTheta = None

# Variation of the angle in time, angular velocity
# at a specific time, in radians per second
StatePendulumDthetaDt = None
```

Remark: the magnitudes should be represented through appropriate physical *measurement units*, as in the example just provided.

2.1.3 Second Example: Water Surface Simulation

Another more complex example, which we shall study in detail this course, is related to the simulation of waves on the water surface. To simplify, consider the 2D section of a water deposit (see the next image). The height of the water surface level is the main parameter to be simulated and visualized.



The vertical height corresponding to each horizontal position at each time is what should be simulated/visualized. In practice one cannot represent infinite horizontal positions; if, for example, the horizontal dimension measures 10 m, we could represent the height every 10 (horizontal) cm. The corresponding state in Processing would be:

```
# World size, in metres.
WorldSize = 10.0

# Space between two consecutive heights, in metres (.1 m = 10 cm).
delta_x = 0.1

# Size of the array of heights representing the surface wave in this
# 10 m world each time.
ArraySize = int(ceil(WorldSize/delta_x))

# Definition of the heights array.
IndexHeightArray = [0.0]*ArraySize
```

This is not the complete state necessary for the water surface simulation. It will be completed later when discussing the wave equation (a PDE).

Remark: the array dimension is obtained through a formula involving the deposit size and the horizontal sampling (also called *resolution*) - in this case 0.1 m. Using formulas is more flexible. An alternative formulation, where the resolution is obtained from world size and array dimension would be:

```
# World size, in metres.
WorldSize = 10.0

# Array size
ArraySize = 100

# Separation between two consecutive height measures, in metres.
delta_x = WorldSize/ArraySize

# Definition of the heights array.
IndexHeightArray = [None]*ArraySize;
```

Before discussing visualization, let us introduce basic programming with Processing.

2.2 Basic *Processing*

Processing was created to enable quick simulation and visualization. What follows is based on: Reas, C, and Fry, B.: *Getting Started with Processing.* You should download and install Processing first.

2.2.1 Basic Visualization

Example 1

Open Processing and try:

```
# Draw a circle centre (50,50) and radius 80.
ellipse(50, 50, 80, 80);
```

You should have copied the code and pasted it in the *editor*. Press *Run* to visualize the results. Try next:

```
ellipse(30, 40, 20, 30);
```

and

```
ellipse(30, 40, 30, 20);
```

Remark:

- A circle is drawn as an ellipse with two equal axes.
- The origin (of the pixel coordinate system used) is at the top left of the window. Positive values of the first coordinate go to the right, those of the second, downwards.
- Comments (beginning with #) are ignored, and are useful to document the code.

Example 2

The next example is interactive:

```
def setup():
    # Instead of the default window size, use a custom window size.
    # Our (rectangular) window is 480 pixels width and 120 height
    size(480, 120)
    smooth()

def draw():
    if mousePressed:
        # If the mouse button is pressed black (0) is painted
        fill(0)
    else:
        # If the mouse button is not pressed white (255) is painted
        fill(255)
        # What is being painted (either black or white) is a circle centered
        # where the mouse lies and 80 pixel radius

    ellipse(mouseX, mouseY, 80, 80)
```

The mini-programs in Processing are called **sketches**. Open a saved sketch, create a new one, export a sketch.

Drawing Elementary Geometric Shapes

A line segment is defined by its two endpoints; a triangle by its 3 vertices, a rectangle by 4 (thus, 8 values). Note that a square is a specific case of a rectangle. Try, for instance:

```
line(20, 50, 420, 110)
triangle(347, 54, 392, 9, 392, 66)
triangle(158, 55, 290, 91, 290, 112)
quad(158, 55, 199, 14, 392, 66, 351, 107)
```

Parts of an ellipse can be drawn through giving the start and end angles (in radians!). Trying the following examples illustrates the positive sign as well:

```
arc(90, 60, 80, 80, 0, HALF_PI)
arc(190, 60, 80, 80, 0, PI+HALF_PI)
arc(290, 60, 80, 80, PI, TWO_PI+HALF_PI)
arc(390, 60, 80, 80, QUARTER_PI, PI+QUARTER_PI)
```

The function radians(·) transforms degrees into radians (e.g.: $radians(90) = \frac{\pi}{2}$).

Less Basic Drawing

Find a tutorial with examples to:

- control the drawing order (paint a rectangle on an ellipse and viceversa)
- smooth lines (function `smooth( )`, already used; `noSmooth( )` will disable smooth)
- define the thickness of the stroke (`strokeWeight( )`) and more advanced functions

```
size(480, 120)
smooth()
ellipse(75, 60, 90, 90)
# stroke thickness 8 pixels
strokeWeight(8)
ellipse(175, 60, 90, 90)
ellipse(279, 60, 90, 90)
# stroke thickness 20 pixels
strokeWeight(20)
ellipse(389, 60, 90, 90)
```

- define colour (grey level or RGB)

```
size(480, 120)
smooth()
# Black
background(0)
# Light grey
fill(204)
# Circle in light grey
ellipse(132, 82, 200, 200)
# Middle grey
fill(153)
# Circle in middle grey
ellipse(228, -16, 200, 200)
# Dark grey
fill(102)
# Circle in dark grey
ellipse(268, 118, 200, 200)
```

- Using colour:

```
size(480, 120)
noStroke()
smooth()
background(0, 26, 51)
fill(255, 0, 0)
# Red circle
ellipse(132, 82, 200, 200)
fill(0, 255, 0)
# Green circle
ellipse(228, -16, 200, 200)
fill(0, 0, 255)
# Blue circle
ellipse(268, 118, 200, 200)
```

By default the most recently painted circle covers the previous one.

2.2.2 Variables (and their Visualization)

Basic Aspects

To declare and assign values to variables

```
# x is declared as a variable without value
int x = None
# Assign a value to x
x = 12
```

More concisely

```
# x is declared as an variable of value 12
x = 12
```

There are default variables, which can be reused. The following code paints circles and lines depending on the window size:

```
size(480, 120)
smooth()
# segment from (0,0) to (480, 120)
line(0, 0, width, height)
# segment from (480, 0) to (0, 120)
line(width, 0, 0, height)
# circle centred at  (240, 60) of radius 30
ellipse(width/2, height/2, 30, 30)
```

We can perform operations (+,-,*,/) with the variables: see the example

```
size(480, 120)
x = 25
h = 20
y = 25
# upper
rect(x, y, 300, h)
x = x + 100
# middle
rect(x, y + h, 300, h)
x = x - 250
# lower
rect(x, y + h*2, 300, h)
```

Declaration and assignment can use operations:

```
# Assigning 24 to  x
x = 4 + 4 * 5;
```

Loops (for)

One can use loops for repetitive tasks, for instance the following code, which paints lines:

```
size(480, 120)
smooth()
strokeWeight(8)
line(20, 40, 80, 80)
line(80, 40, 140, 80)
line(140, 40, 200, 80)
line(200, 40, 260, 80)
line(260, 40, 320, 80)
line(320, 40, 380, 80)
```

can be replaced by

```
size(480, 120)
smooth()
strokeWeight(8)
for i in range(20,350,60):
    line(i, 40, i + 60, 80)
```

After the definition of the “for” we have a colon (;) and the commands that will be executed within the loop are indented with four spaces (“ ”). The “range(init, end, step)” function allows for controlling the iteration. Take into account that the end and the step have to be controlled carefully, as range(0,6,2) will produce the indices ”0,2,4”, whereas range(0,7,2) will produce the indices (0,2,4,6).

2.2.3 Two Basic Methods: *Setup* and *Draw*

It is easy to see how both methods work through a simple example, using a function that writes on the console, `print()`:

```
def setup():
    print(              )

def draw():
    println(             )
```

Alternating quickly *Run* and *Stop*, one sees that `I'm starting` is written once, while `I'm running` is written until we stop. Within *setup* one usually puts things that are defined just once such as the window

size; within *draw* things such as the user's input, as the program should normally watch continuously for it.

The second example given above can be better understood now: at *set up* the window and the fixed `smooth` function were defined; at *draw* circles are painted, centered where the user had the mouse and colour determined by the mouse button being pressed or not. The example with comments follows:

```
def setup():
    # Definition of the (rectangular) window size. Width 480 pixels, Height 120.
    size(480, 120)
    smooth()

def draw():
    if mousePressed:
        // If the mouse button is pressed black (0) is painted
        fill(0)
    else:
        # If the mouse button is not pressed white (255) is painted
        fill(255)
    # What is being painted (either black or white) is a circle centered
    # where the mouse lies and 80 pixel radius
    ellipse(mouseX, mouseY, 80, 80)
```

2.3 Visualizing Pendulum and Waves

2.3.1 Pendulum Visualization

First, we transform the *physical* (world) coordinate system to the screen system, which uses pixels. If a 2 m world is visualized on a 300x300 screen, then the heading of the program could be:

```
WindowWidthHeight = 300
WorldSize = 2.0
PixelsPerMeter = float(WindowWidthHeight/WorldSize)
OriginPixelsX = 0.5*float(WindowWidthHeight)
OriginPixelsY = 0.25*float(WindowWidthHeight)
```

Withini *setup* some values can be modified. Specifically, the mutable variables that have been defined in the heading. Many mutable objects exist (lists, sets, arrays, dicts) for storing data.

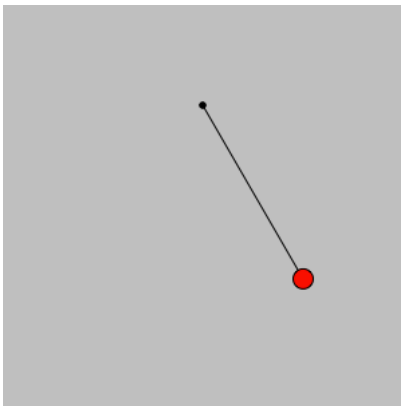
```
GlobalInmutableVariable = 10
GlobalMutableVariable = [10]

def setup():
    LocalVariable = [10,20]

    # Testing if variables are modifiable
    # Modifying global inmutable variables will raise errors
    GlobalInmutableVariable = 5 # This will raise an error

    # Modifying global mutable variables will not raise errors
    GlobalMutableVariable[0] = 5 # Will not raise an error

    # Modifying any local variables will not raise errors
    LocalVariable[1] = 5 # Will not raise an error
    LocalVariable = 5 # Will not raise an error
```

Within *draw* we call a function DrawState to paint some of the state elements: the arm, the pendulum pivot and bob; at each time we need the angle as well. This is an auxiliary function defined ad-hoc for this purpose, and is not a reserved word. The code:

```
# The DrawState function assumes that the simulation coordinate
# system is measured and the pendulum pivot is at the origin. Draw
# the arm, the pivot and the bob
def DrawState():
    # Compute the arm end.
    ArmEndX = PixelsPerMeter*PendulumLength*sin(StatePendulumTheta)
    ArmEndY = PixelsPerMeter*PendulumLength*cos(StatePendulumTheta)

    # Draw the arm.
    strokeWeight( 1.0 )
    line(0.0, 0.0, ArmEndX, ArmEndY)

```

```
    # Draw the pivot.
    fill( 0.0 )
    ellipse(0.0, 0.0, 0.03*PixelsPerMeter, 0.03*PixelsPerMeter)

    # Draw the bob.
    fill( 1.0, 0.0, 0.0 )
    ellipse(ArmEndX, ArmEndY, 0.1*PixelsPerMeter, 0.1*PixelsPerMeter)
```

The following image shows the visualization assuming the angle is $\pi/6$. This way one can test whether the visualization code is working. In order to simulate, we need to introduce the solution of the corresponding DE.

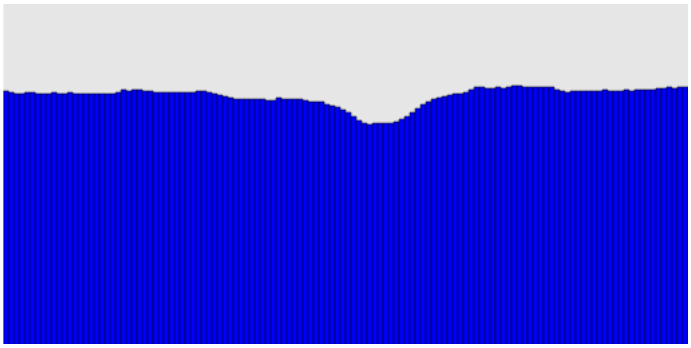


2.3.2 Drawing Surface Waves

The next example, simulated later, is the visualization of the heights 1D field for the surface waves. Now we need a more complex DrawState function which paints blue rectangles whose width is the resolution, and whose height is that of the current state; they should be painted one next to each other filling the whole deposit. At the beginning, the suitable world to screen coordinate system transformations are defined. The code can be:

```
def DrawState():
    OffsetY = 0.5 * ( float )WindowHeight
    for i in range(ArraySize):
        SimX = delta_x * float(i)
        PixelsX = float(i * PixelsPerCell)
        SimY = IndexHeight[i]
        PixelsY = SimY*float(PixelsPerCell)/delta_x
        PixelsMinY = OffsetY - PixelsY
        PixelsHeight = float(WindowHeight)- PixelsMinY
        fill( 0.0, 0.0, 1.0 )
        rect( PixelsX, OffsetY - PixelsY, PixelsPerCell, PixelsHeight )
```

The visualization, if correctly programmed, would be as in the following image. To make it move, we need the simulation, the solution of the DE.



2.4 Other Examples, Coding Style

More complex states can be illustrated through the following example. There are a number of particles, with several attributes each, such as position, mass, speed, etc. The state at a time depends usually on

the values at previous times, and thus the latter should be part of the state, to facilitate the simulation. The following code exemplifies this:

```
# Max number of particles.
MAX_NUM_PARTICLES = 1024
ArraySize = MAX_NUM_PARTICLES

# Arrays of data of each particle. There is an array for each attribute,
# separating eventually the vector components.
# This illustrates a simulation strategy:
# structs of arrays, rather than arrays of structs.
StateMass     = [None]*ArraySize # Or [0.0]*ArraySize
StateRadius   = [None]*ArraySize # Or [0.0]*ArraySize
StatePosX     = [None]*ArraySize # Or [0.0]*ArraySize
StatePosY     = [None]*ArraySize # Or [0.0]*ArraySize
IndexVelocityX     = [None]*ArraySize # Or [0.0]*ArraySize
IndexVelocityY     = [None]*ArraySize # Or [0.0]*ArraySize
StatePrevPosX = [None]*ArraySize # Or [0.0]*ArraySize
StatePrevPosY = [None]*ArraySize # Or [0.0]*ArraySize
```

The following could be the full state of the surface waves example:

```
# World size, in metres.
WorldSize = 10.0

# Space between two consecutive heights, in metres
delta_x = 0.1 # (.1 m = 10 cm)

# Array size of the heights representing the surface wave
# of the 10 m. world at some time.
ArraySize = int(ceil( WorldSize/delta_x))

# Defining the heights array.
IndexHeight      = [None]*ArraySize # Or [0.0]*ArraySize
StateDheightDt   = [None]*ArraySize # Or [0.0]*ArraySize
StatePressure    = [None]*ArraySize # Or [0.0]*ArraySize
StateFloorHeight = [None]*ArraySize # Or [0.0]*ArraySize
StatePrevHeight  = [None]*ArraySize # Or [0.0]*ArraySize
```

2.4.1 Coding Conventions Used

Coding conventions are important to facilitate use and re-use of code. Remark that in each example so far always:

- the size of arrays is ArraySize
- the names of the state variables have the prefix State,
- the size of the world is WorldSize,
- the horizontal resolution is delta_x,

and so on.

2.4.2 Abstracting the State

Another abstraction level can come from putting all the magnitudes within a *state* at each time; an index will denote each magnitude used. Thus each magnitude could be directly accessed if required. This macro-vector with all the magnitudes will be always called **State Vector**. See the particles example:

```
# Define the elements in the state
ArraySize = 1024
NumStateElements = 8

# Define the state: one element for the time and ArraySize
# elements for the rest of the elements
State = [0.0] + [[0.0]*ArraySize]*NumStateElements

# Indices to access the State variable
IndexTime = 0
IndexMass = 1
```

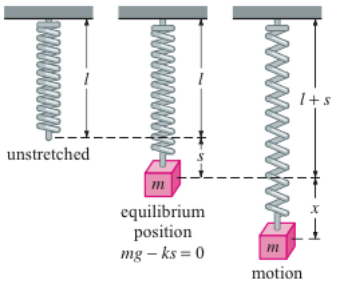
```
IndexRadius = 2
IndexPosX = 3
IndexPosY = 4
IndexVelX = 5
IndexVelY = 6
IndexPrevPosX = 7
IndexPrevPosY = 8

# To access the position x of particle i:
xPos = State[IndexPosX][i]
```

Chapter 3

The Elastic Spring: One-step Methods

3.1 A Physical Model of the Spring

In this chapter we focus on simulating the dynamical behaviour of a vertical spring. We assume that the whole weight is concentrated at the lower end. The simple physical model we consider assumes that the spring has perfect *elasticity*, i.e., the recovery force is proportional to the displacement from the equilibrium position, represented as x, i.e., $F = -Cx$, where F is the recovery force, C is the *elasticity constant*; and note the minus sign which means that the force goes in the opposite direction of the displacement. See the following figure:



Using Newton's second law, $F = ma$, the model can be represented as:

$$\frac{F(t)}{m} = a(t) = \frac{d^2x}{dt^2} = -\frac{C}{m}x$$

This is a second order ODE in the displacement x. The initial conditions of the corresponding IVP are the initial position and velocity. **Remark**: a second order ODE requires *two* inital conditions.

Exercise 9
Consider a vertical spring with a weight, as in the previous image. Determine the IVP of the vertical movement of the weight knowing that a 2 Kg mass extends the spring 6 cm; that at $t = 0$ the mass is 8 cm below the equilibrium position, with a 16 cm/s initial upwards speed.

3.2 Numerical Solution of the Spring ODE through Euler

3.2.1 From a Second Order ODE to a First Order System

A second order ODE, such as the one obtained for the elastic string, can be transformed into a *system of two first order ODEs*, in this case just expressing acceleration as derivative of velocity, resulting thus:

$$\frac{dx(t)}{dt} = v(t)$$

$$a(t) = \frac{dv(t)}{dt} = -\frac{C}{m}x$$

where we see that each equation is first order, as only first derivatives appear in it; although the equations are "coupled".

3.2.2 Euler's Method Applied to the First Order System

In the first order system just obtained for the elastic spring:

$$\frac{dx(t)}{dt} = v(t) \qquad \frac{dv(t)}{dt} = a(t) = -\frac{C}{m}x$$

we approximate each equation using Euler's method, so that we have

$$x_{k+1} = x_k + hv_k$$

$$v_{k+1} = v_k + h(-\frac{C}{m}x_k)$$

This formulation is not suitable for the computation: when computing the first equation the x would be updated to x_{k+1}; in order to apply correctly the second equation, one would have to keep the previous x_k; and starting first with the second equation would lead to a similar issue. Instead, we use $a_k = -\frac{C}{m}x_k$, and the following iterative scheme, considering the order, gives correct results.

Assume that the k step has been computed, i.e., x_k and v_k have been obtained, then

$$a_k = -\frac{C}{m}x_k$$

$$x_{k+1} = x_k + hv_k$$

$$v_{k+1} = v_k + ha_k$$

In the following step we would compute a_{k+1}, x_{k+2} and v_{k+2} and so on. We would use a *for* loop to perform the computations. The initial position x_0 and initial speed v_0 are used to start the iterations, computing from them a_0, x_1 and v_1 as the first iteration.

Exercise 10
Using $h = 0.1$ compute the first 10 steps of the numerical solution to the IVP of the spring initial value, assuming that $m = C = 1$ and that the initial position and velocity are 2 and 0, respectively.

3.3 Coding Euler's Method Applied to the 1D Spring

3.3.1 State and Visualization of the 1D Spring

Following the strategy set in Chapter 2, we define first the *state* and the *visualization* in this subsection, while in the next one we deal with the *simulation*.

Representing the *system state* requires the **constants** C and m appearing in the equations; as **variables** we need *time*, *position* and *speed*: the acceleration, which also appears in the equation, does not need an explicit representation in this case. On the other hand, we only use the *displacement from the equilibrium*, from which the actual position can be obtained. Thus, using Processing one would write:

```
# Constants
Stiffness = 5.0
BobMass = 0.5

# Define the state
State = [0.0, 0.0, 0.0]

# Indices to access the state
IndexTime = 0
IndexPosition = 1
IndexVelocity = 2
```

In another block the *visualization parameters* are defined:

```
WindowWidthHeight = 300
WorldSize = 2.0
PixelsPerMeter = float(WindowWidthHeight)/WorldSize
OriginPixelsX = 0.5*float(WindowWidthHeight)
OriginPixelsY = 0.5*float(WindowWidthHeight)
```

The usual Processing blocks, *setup* and *draw*, can be as:

```
def setup():
    # Create the initial state.
    State[IndexTime] = 0.0
    State[IndexPosition] = 0.65
    State[IndexVelocity] = 0.0

    # Establish normalized colours.
    colorMode( RGB, 1.0 )

    # Establish colour and width of stroke.
    stroke( 0.0 )
    strokeWeight( 0.01 )

    # Create the window size, and establish the transformation of variables.
    size(WindowWidthHeight, WindowWidthHeight)

# Function to draw the state. Remark that only position is visualized.
def DrawState():
    # Compute the end of the arm.
    float SpringEndX = PixelsPerMeter * State[IndexPosition]

    # Draw the spring.
    strokeWeight( 1.0 )
    line( 0.0, 0.0, SpringEndX, 0.0 )

    # Draw the pivot of the string.
    fill( 0.0 )
    ellipse( 0.0, 0.0,  PixelsPerMeter * 0.03, PixelsPerMeter * 0.03 )

    # Draw the weight at the end of the spring.
    fill( 1.0, 0.0, 0.0 )
    ellipse( SpringEndX, 0.0, PixelsPerMeter * 0.1, PixelsPerMeter * 0.1 )

# The Processing Draw function is called every time the screen is refreshed.
def draw():
    background( 0.75 )

    # Translate to the origin.
    translate( OriginPixelsX, OriginPixelsY )

    # Draw the simulation
    DrawState()
```

Then, check that the visualization is correct.

3.3.2 The Time Step Function

The simulation requires to code the numerical solution of the equation. For that, we add a *TimeStep* function within the *DrawState* function, based on Euler's method. As this function is within Draw, a new computation will be carried out for each screen refreshment, as required by Euler's method, which is iterative, and the TimeStep must be a loop. An additional variable needed for the TimeStep is the time increment $\Delta t = h$, which we denote as *delta_t*, which will be the usual notation.

```
# TimeStep function.
def TimeStep(delta_t):
    # Compute acceleration from current position.
    A = (-Stiffness/BobMass)*State[IndexPosition]

    # Update position from current velocity.
    State[IndexPosition] += delta_t*State[IndexVelocity]

    # Update velocity from current acceleration.
    State[IndexVelocity] += delta_t*A

    # Update time.
    State[IndexTime] += delta_t
```

As indicated, we call *TimeStep* within *draw*. If the frame rate is 24, the consistent time increment is 1/24 second, as shown below:

```
def draw():
    # Time increase.
```

```
    TimeStep( 1.0/24.0 )

    # Constant colour background.
    background( 0.75 )

    # Translate to origin.
    translate( OriginPixelsX, OriginPixelsY )

    # Draw simulation
    DrawState()
```

This visualization, based on the numerical solution of the ODE using the Euler's method, shows that the solution we obtain does not seem to be correct, as the spring end moves quicker and quicker and finally leaves the window. The numerical solution and simulation are *unstable*. A better numerical solution obtained through a different method is needed.

3.3.3 Comparing to the Exact Solution

The exact solution of the spring ODE is known, as seen later, and it can be drawn at the same time as the numerical approximation, which allows us for visual *debugging*. Indeed, we add the formula of the exact solution within *DrawState*, painting it at a different position with a different colour, as follows:

```
def DrawState():
    # Compute the end of the arm.
    SpringEndX = PixelsPerMeter*State[IndexPosition]

    # Compute the CORRECT position.
    sqrtKoverM = sqrt( Stiffness / BobMass )
    x0 = InitState[IndexPosition]
    v0 = InitState[IndexVelocity]
    t = State[IndexTime]
    CorrectPositionX = (x0*cos(sqrtKoverM*t)) +
    ((v0/sqrtKoverM)*sin(sqrtKoverM + t))

    # Compute the end of the arm corresponding to the correct position
    CorrectEndX = PixelsPerMeter*CorrectPositionX

    # Draw the spring.
    strokeWeight(1.0)
    line(0.0, 0.0, SpringEndX, 0.0)

    # Draw the spring pivot.
    fill(0.0)
    ellipse(0.0, 0.0, 0.03*PixelsPerMeter, 0.03*PixelsPerMeter)

    # Draw the weight at the end of the spring.
    fill( 1.0, 0.0, 0.0 )
    ellipse(SpringEndX, 0.0, 0.1*PixelsPerMeter, 0.1*PixelsPerMeter)

    # Draw the "correct" weight in blue.
    fill( 0.0, 0.0, 1.0 )
    ellipse(CorrectEndX, -PixelsPerMeter * 0.25,
            PixelsPerMeter * 0.1,
            PixelsPerMeter * 0.1 )
```

Reset

It is useful to introduce in the code a *reset* possibility. The following code resets when the r key (ascii code 124) is released: the code copies the initial state into the current one, so that the simulation begins again starting with the initial values.

```

# Reset function. When the "r" ke is released while drawing,
# it resets the state to the initial state
def keyReleased():
    if key == 114:
        # Reset to initial state
        State[IndexTime] = 0.0
        State[IndexPosition] = 0.65
        State[IndexVelocity] = 0.0
```

3.4 Higher Order, Implicit, Multistep Methods

The simulation of the elastic spring using the (very simple) Euler's numerical method shows important limitations of this method. More complex methods are needed for a better approximation. Two improvement strategies are:

1. To interpolate several approximations to compute a further one. The best known methods that follow this strategy for ODEs go under the **Runge-Kutta (RK) methods** name. The order of precision is higher.

2. To compute the approximation at a step using future steps as well. This leads to more complex numerical schemes, and the methods are called **implicit**.

Both strategies fall within the category of **one-step methods**. There are **multistep methods** as well.

We illustrate those methods and their effects on the elastic spring simulation.

3.5 A Semi-implicit Method on the Spring: Euler-Cromer

Euler's numerical scheme for the elastic spring:

$$a_k = -\frac{C}{m}x_k$$

$$x_{k+1} = x_k + hv_k$$

$$v_{k+1} = v_k + ha_k$$

can be turned into semi-implicit by updating *first* the velocity, and *then* the position, i.e.,

$$a_k = -\frac{C}{m}x_k$$

$$v_{k+1} = v_k + ha_k$$

$$x_{k+1} = x_k + hv_{k+1}$$

This is partly implicit as we use the *next* (future) velocity to compute the *current* position. It is partly explicit in the computation of the next velocity from the current acceleration.

This scheme is called Euler-Cromer, semi-implicit Euler, simplectic Euler, semi-explicit Euler, or Newton-Størmer-Verlet (NSV).

The improvement is remarkable: the simulation becomes stable, showing the strength of the implicit strategies.

Changing the code is very simple as well: the implementation just exchanges two lines of code (in the TimeStep function):

```
# Euler-Cromer TimeStep function.
def TimeStep(delta_t):
    # Compute acceleration from current position.
    A = ( -Stiffness / BobMass ) * State[IndexPosition]

    # Update velocity from current acceleration.
    State[IndexVelocity] += delta_t * A

    # Update position from updated velocity.
    State[IndexPosition] += delta_t * State[IndexVelocity]

    # Update time.
    State[IndexTime] += delta_t
```

The improvement is huge, but still the numerical solution simulated diverges from the exact one as time increases. Other methods improve the approximation.

3.6 Higher Order Methods: Runge-Kutta

Consider again the first order ODE $\frac{dx}{dt} = x'(t) = f(t,x)$, $\Delta t = h$ and $kh = t_k$. Recall that Euler's method is given by the iteration:

$$x_{k+1} = x_k + hf(t_k, x_k)$$

Remark that this formula consists of the first two terms of Taylor's expansion for x_{k+1}. Let us remember that we denoted $\Delta t = h$, $kh = t_k$, and x_k the approximate solution at t_k. Thus, $x_{k+1} = x((k+1)h) = x(kh+h) = x(t_k+h)$. If we expand $x(t_k+h)$ using Taylor's formula, recalling that $x' = f$ and leave only the first two terms, we have $x_{k+1} = x(t_k+h) = x(t_k) + hf(t_k,x_k) = x_k + hf(t_k,x_k)$. Some of the $=$ signs used should have been $\simeq$, in order to be more precise.

Higher order methods, such as Runge-Kutta are based on:

$$x_{k+1} = x_k + h\phi(t_k, x_k)$$

where $\phi(t,x)$ is an interpolation of several values of f, two in second order methods, four in fourth order methods. And we use Taylor's expansion to obtain them, as seen next.

3.6.1 Runge-Kutta 2 (Heun) Method

For RK2 we set $\phi(t,x) = c_2 f(t,x) + c_3 f(t + c_1 h, x + c_1 f(t,x))$, and to obtain the constants c_1, c_2 and c_3, we use Taylor's formula. Indeed, using Taylor's expansion of $x(t+h)$ up to the third order we get:

$$\frac{x(t+h) - x(t)}{h} = x'(t) + \frac{1}{2}hx''(t) + \frac{1}{6}h^2 x'''(t) + 0(h^3)$$

Note that $x'(t) = f(t,x)$. We can write $x''(t)$ as:

$$x''(t) = \frac{df}{dt} = \frac{\partial f}{\partial t} + \frac{\partial f}{\partial x}\frac{dx}{dt} = f_t + f_x f$$

where we use a common terminology for partial derivatives $\frac{\partial f}{\partial t} = f_t$ and $\frac{\partial f}{\partial x} = f_x$ and do not indicate explicitly where the derivatives are evaluated. $x'''(t)$ can be expressed in a similar way, and substituting in Taylor's expansion we have:

$$\frac{x(t+h) - x(t)}{h} = f + \frac{1}{2}h(f_t + f_x f) + \frac{1}{6}h^2(f^2 f_{xx} + 2ff_{tx} + f_{tt} + f_t f_x + f f_x^2) + 0(h^3)$$

Now, expanding $\phi(t, x(t))$ using Taylor's formula:

$$\phi(t,x(t)) = (c_2+c_3)f + hc_1c_3(f_t + f_x f) + \frac{1}{2}h^2 c_1^2 c_3(f^2 f_{xx} + 2ff_{tx} + f_{tt} + f_t f_x + f f_x^2) + 0(h^3)$$

And then,

$$\frac{x(t+h) - x(t)}{h} - \phi(t,x(t)) = (1 - c_2 - c_3)f + h(\frac{1}{2} - c_1c_3)(f_t + f_x f) +$$

$$+\frac{h^2}{2}(\frac{1}{3} - c_1^2 c_3)(f^2 f_{xx} + 2ff_{tx} + f_{tt}) + \frac{h^2}{6}(f_t f_x + f f_x^2) + 0(h^3)$$

If we make $1 = c_2 + c_3$ and $\frac{1}{2} = c_1 c_3$ the two first terms are 0, so that the (local discretization) error is $0(h^2)$, i.e., *second order* - that is why we say RK2 is a second order method, unlike Euler's, which is first order.

The equations $1 = c_2 + c_3$, $\frac{1}{2} = c_1 c_3$ have infinite solutions. One of them, $c_2 = c_3 = \frac{1}{2}$, $c_1 = 1$, results into **Heun's method** or **RK2 method**, namely:

$$x_{k+1} = x_k + \frac{h}{2}\left[f(t_k, x_k) + f(t_{k+1}, x_k + hf(t_k, x_k))\right]$$

Remark: If we define

$$F_{1,k} = f(t_k, x_k) \qquad F_{2,k} = f(t_{k+1}, x_k + hF_{1,k})$$

Then, the RK2 method can be expressed in the more compact way:

$$x_{k+1} = x_k + \frac{h}{2}(F_{1,k} + F_{2,k})$$

EXERCISE 11
The exact solution of the IVP $x'(t) = x$, $x(0) = 1$ is e^t.

- *Using Heun's method compute the approximate value of e^1, using $\frac{1}{4}$, $\frac{1}{8}$, or $\frac{1}{16}$ for $h = \Delta t$. Remark that the number of iterations needed will be different according to the value of Δt chosen. Note that we did the same thing in a previous exercise using Euler's method instead of Heun's.*
- *Compute the absolute and relative errors of Heun's method for the different values of Δt and compare them.*
- *Compare the errors when using Euler's and Heun's methods.*

3.6.2 Runge-Kutta 4 Method

A derivation using Taylor similar to that of Heun's, but more complex, leads to the RK4 method, which can be expressed in a compact way as:

$$x_{k+1} = x_k + \frac{h}{6}(F_{1,k} + 2F_{2,k} + 2F_{3,k} + F_{4,k})$$

where we use the terminology

$$F_{1,k} = f(t_k, x_k) \qquad F_{2,k} = f(t_k + \frac{h}{2}, x_k + \frac{h}{2}F_{1,k})$$

$$F_{3,k} = f(t_k + \frac{h}{2}, x_k + \frac{h}{2}F_{2,k}) \qquad F_{4,k} = f(t_{k+1}, x_k + hF_{3,k})$$

following a strategy similar as that in the previous subsection.

EXERCISE 12
Repeat the previous exercise using now RK4, computing the approximate value of e^1 for different values of Δt, and compare the errors with those previously obtained.

EXERCISE 13
Consider the IVP $x'(t) = x^2(t) + 2t - t^4$, $x(0) = 0$, whose solution is t^2 .

- *Compute the approximate values of the solution at times $t = 0.1, 0.2, ..., 1$ (i.e., using $\Delta t = h = 0.1$) using RK2 and RK4.*
- *Compare the values obtained through Euler's, Heun's and RK4 methods to the exact values at those times.*
- *Determine the errors in the interval $t \in [0, 1]$.*

3.7 Simulation of the Elastic Spring with RK2 i RK4

3.7.1 Simulation of the Elastic Spring with RK2

Recall that the elastic spring second order ODE becomes a first order system as:

$$\frac{dx}{dt} = v$$

$$\frac{dv}{dt} = -\frac{C}{m}x$$

which can be expressed in vector terms as:

$$\begin{bmatrix} \frac{dx}{dt} \\ \frac{dv}{dt} \end{bmatrix} = \begin{bmatrix} v \\ -\frac{C}{m}x \end{bmatrix}$$

and Euler's approximation in vector form is:

$$\begin{bmatrix} x_{k+1} \\ v_{k+1} \end{bmatrix} = \begin{bmatrix} x_k \\ v_k \end{bmatrix} + h \begin{bmatrix} v_k \\ -\frac{C}{m}x_k \end{bmatrix}$$

As the RK2 method is:

$$x_{k+1} = x_k + \frac{h}{2}\left[f(t_k, x_k) + f(t_{k+1}, x_k + hf(t_k, x_k))\right]$$

we can apply it to the elastic spring system in vector form:

$$\begin{bmatrix} x_{k+1} \\ v_{k+1} \end{bmatrix} = \begin{bmatrix} x_k \\ v_k \end{bmatrix} + \frac{h}{2}\left\{ \begin{bmatrix} v_k \\ -\frac{C}{m}x_k \end{bmatrix} + \begin{bmatrix} v_k + ha_k \\ -\frac{C}{m}(x_k + hv_k) \end{bmatrix} \right\}$$

Remark that RK2 is $x_{k+1} = x_k + \frac{h}{2}\left[f(t_k, x_k) + f(t_{k+1}, x_k + hf(t_k, x_k))\right]$ where $x_k + hf(t_k, x_k)$ is the value computed through Euler's method. We can use this to compute the next step without using the vector form, but re-using Euler's computations, namely taking into account that for the elastic spring f has got two components, v and $a = -\frac{C}{m}x$. Thus $f(t_k, x_k)$ is composed by:

$$v^*_{1,k} = v_k \qquad a^*_{1,k} = -\frac{C}{m}x_k$$

and these values can be used to compute $f(t_k, x_k) + f(t_{k+1}, x_k + hf(t_k, x_k))$

$$v^*_{2,k} = v_k + ha^*_{1,k} \qquad a^*_{2,k} = -\frac{C}{m}(x_k + htv^*_{1,k})$$

Thus, the updated x_{k+1} and v_{k+1} will be:

$$x_{k+1} = x_k + \frac{h}{2}(v^*_{1,k} + v^*_{2,k}) \qquad v_{k+1} = v_k + \frac{h}{2}(a^*_{1,k} + a^*_{2,k})$$

This way it is easier to re-use the code from Euler's method. On the other hand, the notation $v^*_{1,k}, a^*_{1,k}, v^*_{2,k}, a^*_{2,k}$ corresponds to that of $F_{1,k}$, $F_{2,k}$ used above, at the same time avoiding undesired updates.

The RK2 code is thus:

```
# TimeStep function.
def TimeStepRK2(delta_t):
    # Position - compute F1
    F1_position = State[IndexVelocity]

    # Velocity - compute F1
    F1_velocity = (-Stiffness/BobMass)*State[IndexPosition]

    # Position - Compute Euler estimate
    x_hat = State[IndexPosition] + (delta_t*F1_position)

    # Velocity - Compute Euler estimate
    v_hat = State[IndexVelocity] + (delta_t*F1_velocity)

    # Position - compute F2
    F2_position = v_hat

    # Velocity - compute F2
    F2_velocity = (-Stiffness/BobMass)*x_hat

    # Position - Update (RK2)
    State[IndexPosition] += (delta_t/2.0)*(F1_position + F2_position)

    # Velocity - Update (RK2)
    State[IndexVelocity] += (delta_t/2.0)*(F1_velocity + F2_velocity)

    # update time.
    State[IndexTime] += delta_t
```

RK2 simulation improves the approximations obtained with Euler. Nevertheless, it increasingly diverges from the exact solution, as seen in the simulation, and could even become unstable. RK2 is second order, RK4 is fourth order and achieves much better results.

3.7.2 Simulation of the Elastic Spring with RK4

The expressions provided earlier for RK4 lead to the following Processing code, using the same strategy as for RK2:

```
# Time step function.
def TimeStepRK4(delta_t):
    # Position - compute F1
    F1_position = State[IndexVelocity]

    # Velocity - compute F1
    F1_velocity = (-Stiffness/BobMass)*State[IndexPosition]

    # Position - Compute Euler estimate
    x_hat = State[IndexPosition] + (delta_t*F1_position)

    # Velocity - Compute Euler estimate
    v_hat = State[IndexVelocity] + (delta_t*F1_velocity)

    # Position - compute F2
    F2_position = v_hat

    # Velocity - compute F2
    F2_velocity = (-Stiffness/BobMass)*x_hat

    # Position - Compute Euler estimate
    x_hat_2 = State[IndexPosition] + (delta_t/2)*F2_position

    # Velocity - Compute Euler estimate
    v_hat_2 = State[IndexVelocity] + (delta_t/2)*F2_velocity

    # Position - compute F3
    F2_position = v_hat_2

    # Velocity - compute F3
    F2_velocity = (-Stiffness/BobMass)*x_hat_2

    # Position - Compute Euler estimate
    x_hat_3 = State[IndexPosition] + (delta_t)/2*F2_position

    # Velocity - Compute Euler estimate
    v_hat_3 = State[IndexVelocity] + (delta_t/2)*F2_velocity

    # Position - compute F4
    F2_position = v_hat_3

    # Velocity - compute F4
    F2_velocity = (-Stiffness/BobMass)*x_hat_3

    # Position - Update (RK4)
    State[IndexPosition] += (delta_t/6.0)*(F1_position + 2*F2_position + 2*F3_position +
     F4_position)

    # Velocity - Update (RK4)
    State[IndexVelocity] += (delta_t/6.0)*(F1_velocity + 2*F2_velocity + 2*F3_velocity +
     F4_velocity)

    # update time.
    State[IndexTime] += delta_t
```

The improvement shown in the simulation is very significant, and thus RK4 is a very popular method. It has got some (storage) limitations which can become important in simulations with millions of points.

3.8 Supplementary Material

3.8.1 Analytical Solution of the Spring ODE

As a Linear ODE

Remark that the spring ODE $\frac{d^2x}{dt^2} = -\frac{C}{m}x$, without considering the initial conditions, is *linear*, as any linear combination $\mu_1 x_1(t) + \mu_2 x_2(t)$ of two solutions $x_1(t)$ and $x_2(t)$ is a solution too. Thus, the (general) solutions form a vector subspace. Within this subspace of functions, if found, the IVP solution would be that satisfying the initial conditions.

To find the subspace, we try whether for some λ, $x(t) = e^{\lambda t}$ is a solution (this is motivated by the fact that the derivatives of the exponential are exponentials too, as it happens with the function x in

this ODE, whose second derivative results into a multiple of the function x). Substituting in the ODE $\frac{d^2x}{dt^2} = -\frac{C}{m}x$ we obtain

$$\lambda^2 e^{\lambda t} = -\frac{k}{m}e^{\lambda t}$$

and thus

$$\lambda^2 = -\frac{k}{m} \qquad \lambda = i\sqrt{\frac{C}{m}}$$

so that

$$x(t) = e^{i\sqrt{\frac{k}{m}}t} = \cos\sqrt{\frac{C}{m}}t + i\sin\sqrt{\frac{C}{m}}t$$

is a solution, which has an imaginary part, and thus it has not a physical meaning. But both the real (cosine) and imaginary (sine) parts are solutions with a physical meaning. The general solution with physical meaning of the ODE can be represented as its linear combination:

$$x(t) = C_1\sin\sqrt{\frac{C}{m}}t + C_2\cos\sqrt{\frac{C}{m}}t$$

To find the IVP solution given specific initial conditions we set $t = 0$ in the general solution:

$$x(0) = x_0 = C_1\sin\sqrt{\frac{C}{m}}0 + C_2\cos\sqrt{\frac{C}{m}}0 = 0C_1 + 1C_2$$

and thus $x_0 = C_2$. To obtain C_1 we derive the general solution $x(t)$:

$$x'(t) = C_1\sqrt{\frac{C}{m}}\cos\sqrt{\frac{C}{m}}t - C_2\sqrt{\frac{k}{m}}\sin\sqrt{\frac{C}{m}}t$$

As $x'(t) = v(t)$ and setting $t = 0$:

$$v(0) = v_0 = C_1\sqrt{\frac{C}{m}}\cos\sqrt{\frac{C}{m}}0 - C_2\sqrt{\frac{C}{m}}\sin\sqrt{\frac{C}{m}}0$$

hence $v_0 = C_1\sqrt{\frac{C}{m}}$, i $\sqrt{\frac{m}{C}}v_0 = C_1$.

The solution corresponding to the initial position x_0 and velocity v_0 is thus:

$$x(t) = \sqrt{\frac{m}{C}}v_0\sin\sqrt{\frac{C}{m}}t + x_0\cos\sqrt{\frac{C}{m}}t$$

(Check that, effectively, it is the solution).

Through Separation of Variables

Another strategy to find the analytical solution from the ODE starts with:

$$\frac{dv(t)}{dt}(= a(t)) = -\frac{C}{m}x$$

where we can apply the *chain rule*:

$$\frac{dv(t)}{dt} = \frac{dv(x)}{dx}\frac{dx(t)}{dt} = \frac{dv}{dx}v(= -\frac{C}{m}x)$$

i.e., $\frac{dv}{dx}v = -\frac{C}{m}x$. This way we can separate the v and x variables, as $vdv = -\frac{C}{m}xdx$, and integrating:

$$\frac{v^2}{2} = -\frac{Cx^2}{2m} + C_1$$

which leads to

$$v = \sqrt{C_1 - \frac{Cx^2}{m}}$$

Undoing the change $\frac{dx(t)}{dt} = v(t)$, and separating again the variables x and t:

$$\frac{dx}{\sqrt{C_1 - \frac{Cx^2}{m}}} = dt$$

Through a trigonometric change such as $x = M \sin Nu$ or $x = M \cos Nu$, using $1 - \sin^2 = \cos^2$ and integrating, a general analytical solution is obtained:

$$x(t) = A \sin \sqrt{\frac{C}{m}} t + B \cos \sqrt{\frac{C}{m}} t$$

The solution to the IVP is obtained as previously.

Exercise 14
Obtain the solution of the first exercise of this chapter.

3.8.2 Variants of the Spring Model

Dampening

A more realistic model includes *dampening*, proportional but opposite, to velocity. The physical model is represented by the ODE:

$$\frac{d^2x}{dt^2} = -\frac{C}{m}x - \beta\frac{dx}{dt}$$

where $-\beta\frac{dx}{dt}$ is the (new) dampening term.

The solution is different according to the relationship between C and β. Indeed, making $2\lambda = \beta$ and $\omega^2 = \frac{C}{m}$ the ODE becomes simpler:

$$\frac{d^2x}{dt^2} + 2\lambda\frac{dx}{dt} + \omega^2 x = 0$$

And then $\lambda > \omega$, $\lambda = \omega$, and $\lambda < \omega$ correspond to three cases: *over-*, *critical* and *sub-* **dampening**.

Exercise 15

- *Find the general solution of the previous ODE, trying solutions* $e^{\gamma t}$
- *and find the solution of the IVP*

  $$\frac{d^2x}{dt^2} + 5\frac{dx}{dt} + 4x = 0 \qquad x(0) = 1 \qquad x'(0) = 1$$

  Which type of dampening exists in this case?

Forced Movement

Another variant is to consider an external force acting on the mass: it is a *forced movement*. If this force is vertical, then the system is represented by the ODE:

$$m\frac{d^2x}{dt^2} = -Cx - \beta\frac{dx}{dt} + F(t)$$

where $F(t)$ is the new term representing the external force. Simplifying the ODE in a similar way as in dampening, we get:

$$\frac{d^2x}{dt^2} + 2\lambda\frac{dx}{dt} + \omega^2 x = f(t)$$

For this equation the diference $x_1 - x_2 = z$ of any two solutions, x_1 i x_2 of the ODE, solves the homogeneous ODE, i.e.:

$$\frac{d^2z}{dt^2} + 2\lambda\frac{dz}{dt} + \omega^2 z = 0$$

The homogeneous equation is linear, and applying a similar strategy as earlier for the simple spring model, its general solution can be obtained. The general solution of the non-homogeneous one results from adding the one obtained for the homogeneous and a particular solution of the non-homogeneous one.

Exercise 16
Find the solution of the IVP

$$\frac{1}{5}\frac{d^2x}{dt^2} + 1.2\frac{dx}{dt} + 2x = 5cos4t \qquad x(0) = \frac{1}{2} \qquad x'(0) = 0$$

Which is the type of dampening?

3.8.3 Paradox: A More Sofisticated Numerical Method with Worse Results

We saw that Euler-Cromer improves over Euler in the case of the elastic spring. It is worth trying the method called **mid-point**. This method consists in reaching the next $t + \Delta t$ through computing first the approximate solution at $t + \frac{\Delta t}{2}$ and using this to compute at $t + \Delta t$. It is not just using Euler with $\frac{\Delta t}{2}$ twice. It is an intermediate approximation to the next step $t + \Delta t$. Indeed, the two steps from t to $t + \Delta t$ through the mid-point method are:

$$x(t + \frac{\Delta t}{2}) = x(t) + \frac{\Delta t}{2}v(t)$$

$$v(t + \Delta t) = v(t) + \Delta t a(t)$$

$$x(t + \Delta t) = x(t + \frac{\Delta t}{2}) + \frac{\Delta t}{2}v(t + \Delta t)$$

Remark that speed is only updated once, while position twice. The corresponding iterative scheme is:

$$a_k = -\frac{C}{m}x_k$$

$$x_{k+\frac{1}{2}} = x_k + \frac{h}{2}v_k$$

$$v_{k+1} = v_k + ha_k$$

$$x_{k+1} = x_{k+\frac{1}{2}} + \frac{h}{2}v_{k+1}$$

And the corresponding program in Processing needs a different TimeStep function, namely:

```
# TimeStep Function
def TimeStep(delta_t):
    # Update position half-way from current speed
    State[IndexPosition] += ( delta_t/2.0 ) * State[IndexVelocity]

    # Update speed from acceleration
    State[IndexVelocity] += delta_t * (-Stiffness/BobMass)*State[IndexPosition]

    # Update position the next half-way
    State[IndexPosition] += ( delta_t/2.0 ) * State[IndexVelocity]

    # Update time
    State[IndexTime] += delta_t
```

The paradox appears when visualising the simulation, as there is no improvement with respect to using Euler, despite making more computations. More work does not mean better results. Indeed, the mid-point method is worse than Euler-Cromer.

Chapter 4

Advanced Methods, Convergence, Stability

4.1 Multi-step Methods, Polynomial Interpolation

Other advanced methods to numerically solve ODEs are **multi-step**. Instead of using *one* value to compute the next step, as we did so far, these methods use *several* values. Notions of (polynomial) interpolation are needed, which we introduce next.

4.1.1 Polynomial Interpolation

Given a set of *nodes* $t_i, i = 0, \ldots, k$, with corresponding values $x_i, i = 0, \ldots, k$, *interpolating* them means finding a function g, such that $g(t_i) = x_i, i = 0, \ldots, k$. If the function g we look for is a polynomial, we say it is a *polynomial interpolation*. Polynomial interpolation is based on the following

THEOREM 4.1
Given a set of $k+1$ different nodes $t_i, i = 0, \ldots, k$, with corresponding values $x_i, i = 0, \ldots, k$, there exists only one polynomial $p(t)$ of degree k or less, which interpolates the nodes and values, i.e., s.t. $p(t_i) = x_i, i = 0, \ldots, k$. □

This theorem can be proved using the $k+1$ Lagrange polynomials of degree k

$$l_i(t) = \prod_{j=0, j\neq i}^{k} \frac{t - t_j}{t_i - t_j} \qquad i = 0, \ldots, k$$

which satisfy $l_i(t_j) = \delta_{ij}$. Thus, the interpolation polynomial can be written as $p(t) = \sum_{i=0}^{k} l_i(t) x_i$. The error of this interpolation is given by the following

THEOREM 4.2
Given a set of nodes $t_0 < t_1 < \ldots < t_k$, a function f which is $k+1$ continuously differentiable in the interval $[t_0, t_k]$ and $p(t)$ is the unique polynomial of degree k or less s.t. $p(t_i) = f(t_i), i = 0, \ldots, k$, then for any $t \in [t_0, t_k]$

$$f(t) - p(t) = \frac{(t - t_0)(t - t_1)\ldots(t - t_k)}{(k+1)!} f^{(k+1)}(\tau)$$

for some $\tau \in [t_0, t_k]$. □

If the nodes are separated by the same distance h, the equality of the theorem can be transformed into a simpler inequality:

$$|f(t) - p(t)| < Mh^{k+1}$$

where M is an upper bound of the absolute value of the $k+1$ derivative on the interval $[t_0, t_k]$. Thus, the polynomial interpolation has a $k+1$ order error.

4.1.2 Newton's formulation

Newton's formulation is more practical to compute the polynomial interpolation coefficients. Defining $\Delta x_i = x_{i+1} - x_i$ and repeatedly applying these differences we get:

$$\Delta^2 x_0 = \Delta x_1 - \Delta x_0 = x_2 - 2x_1 + x_0$$

$$\Delta^3 x_0 = \Delta^2 x_1 - \Delta^2 x_0 = x_3 - 3x_2 + 3x_1 - x_0$$

$$\vdots$$

$$\Delta^k x_0 = x_k - \begin{pmatrix} k \\ 1 \end{pmatrix} x_{k-1} + \begin{pmatrix} k \\ 2 \end{pmatrix} x_{k-2} - \ldots + (-1)^k x_0$$

Then, the polynomial

$$p_k(t) = x_0 + \frac{t - t_0}{h}\Delta x_0 + \frac{(t-t_0)(t-t_1)}{2h^2}\Delta^2 x_0 + \ldots + \frac{(t-t_0)(t-t_1)\ldots(t-t_k)}{k!h^k}\Delta^k x_0$$

where $h = t_{i+1} - t_i$, verifies that $p_k(t_i) = x_i$ and thus it is the interpolation polynomial corresponding to t_i, x_i. Newton is more often used than Lagrange as it is *incremental*, i.e., adding a new point does not require computing again everything, as in the case of using Lagrange polynomials; with Newton's formulation one can just add a term.

4.1.3 Multi-step Methods

As indicated earlier, multi-step methods use several previous instants to compute the next iteration. Indeed, using the known properties of derivatives and integrals on the prototypical first order ODE $\frac{dx}{dt} = x'(t) = f(t, x)$

$$x(t_{k+1}) - x(t_k) = \int_{t_k}^{t_{k+1}} x'(s)ds = \int_{t_k}^{t_{k+1}} f(s, x(s))ds$$

Then, if we approximate f by means of an interpolation polynomial p:

$$x(t_{k+1}) - x(t_k) = \int_{t_k}^{t_{k+1}} f(s, x(s))ds \approx \int_{t_k}^{t_{k+1}} p(s, x(s))ds$$

The multi-step methods are based on using an interpolation polynomial s.t. $p(t_i) = f_i = f(t_i, x_i)$ for some equally spaced nodes t_i.

If just one node t_k is used, the interpolation polynomial is f_k, and Euler's method reappears. If two nodes are used, for instance t_k and t_{k-1}, computing the interpolation polynomial and integrating, we obtain the 2-step method:

$$x_{k+1} = x_k + \frac{h}{2}(3f_k - f_{k-1})$$

If three nodes t_k, t_{k-1} and t_{k-2} are used:

$$x_{k+1} = x_k + \frac{h}{12}(23f_k - 16f_{k-1} + 5f_{k-2})$$

A 4-step method will derive from using t_k, t_{k-1}, t_{k-2} and t_{k-3} :

$$x_{k+1} = x_k + \frac{h}{24}(55f_k - 59f_{k-1} + 37f_{k-2} - 9f_{k-3})$$

These formulae are known as **Adams-Bashforth methods** of order 2, 3, 4, as the number of steps is the same as the order of the method.

An IVP provides the ODE and the initial value. To start the iterations of a multi-step method, several values are needed. These starting values are computed using a single step method, usually the RK of the same order as the Adam-Bashforth used.

4.1.4 Predictor-Corrector Methods

The predictor-corrector methods improve multi-step methods introducing a sort of implicit orientation. For instance, for a two-step method, instead of using the nodes t_k and t_{k-1}, one would use t_k and t_{k+1}, and through polynomial interpolation we get:

$$x_{k+1} = x_k + \frac{h}{2}(f_k + f_{k+1})$$

which is known as the **second order Adams-Moulton method**.

As x_{k+1} is not known, neither is f_{k+1}. Then, Adam-Bashforth is used to make a *prediction* of x_{k+1}, and Adams-Moulton to *correct* this prediction.

More precisely, the prediction is made through:

$$\widetilde{x}_{k+1} = x_k + \frac{h}{2}(3f_k - f_{k-1})$$

and the correction through:

$$x_{k+1} = x_k + \frac{h}{2}(f_k + \widetilde{f}_{k+1})$$

The fourth-order predictor-corrector method is the one most frequently used, with prediction:

$$\widetilde{x}_{k+1} = x_k + \frac{h}{24}(55f_k - 59f_{k-1} + 37f_{k-2} - 9f_{k-3})$$

and correction:

$$x_{k+1} = x_k + \frac{h}{24}(9\widetilde{f}_{k+1} + 19f_k - 5f_{k-1} + f_{k-2})$$

Exercise 17
Consider the IVP $x'(t) = xt$ and $x(0) = 1$, whose solution is $e^{\frac{t^2}{2}}$.

1. *Compute the exact value of $x(1.5)$.*
2. *Compute the approximate value of $x(1.5)$ using a second-order predictor-corrector and $\Delta t = h = 0.1$. Pay special attention to the initialization, and justify the steps taken.*
3. *Compare the previous result with that obtained with RK2 and the same h. Which is the relative error in each case?*
4. *Repeat 2 and 3 with fourth order methods.*

4.2 Errors, Convergence, Stability

4.2.1 Types of Errors

Quite a few numerical approximations rely on Taylor's formula, which can be expressed as:

$$x(t + \Delta t) = x(t) + x'(t)\Delta t + \frac{x''(t)}{2!}\Delta t^2 + + \frac{x'''(t)}{3!}\Delta t^3 + h.o.t.$$

Euler's method is based on the approximation $x(t + \Delta t) \approx x(t) + x'(t)\Delta t$, and using then the ODE $x'(t) = f(t, x(t))$. Then, only the first two terms of Taylor's expansion are used: the error in this approximation is given by the higher order terms which are left out, which can be expressed as

$$\frac{\Delta t^2}{2}x''(\tau), \quad \tau \in (t, t + \Delta t)$$

This is the error made at each iteration, when going from t to $t + \Delta t$, and is called **local error**. Another type of error is the one made when going from an initial time a, usually 0, where we assume that the initial conditions are known without error, to a later time b. This is called the **global error**: it results from the accumulation of the errors at each iteration.

4.2.2 Local Error for Euler's Method

Recall that $x(t_k)$ is the exact value of the solution at $t_k = t_0 + kh$, and x_k the approximate value obtained numerically. To estimate the local error at an iteration, let us assume that $x(t_k) = x_k$. The error made when computing the next value through Euler is:

$$x(t_{k+1}) - x_{k+1} = x(t_{k+1}) - x(t_k) - hf(t_k, x(t_k)) = h\left[\frac{x(t_{k+1}) - x(t_k)}{h} - f(t_k, x(t_k))\right] = hL(t_k, h)$$

where we denote $L(t,h) = \frac{x(t+h)-x(t)}{h} - f(t, x(t))$. L is known as **local discretization error** - in Euler's method. Indeed it is the difference from the derivative and the approximation used for it.

A bound for L in the interval $[a, b]$ is its maximum value in the interval. As $L(t_k, h) = \frac{h^2}{2}x''(\tau)$ we can write :

$$L(h) = \max_{a \leq t \leq b} |L(t,h)| \leq \frac{h}{2}M$$

where M is a bound of the second derivative in the interval. This bound confirms an intuition, that decreasing h, the error L decreases (at the price of a higher computational cost). It is said that $L = 0(h)$, as L tends to zero at the same pace as h.

4.2.3 Global Error of Euler's Method and Convergence

Let us assume that $nh = b - a$, so that n iterations are needed to obtain the approximate value of $x(b)$. A local error is accumulated at each iteration. Denoting each local error $x(t_k) - x_k$ by e_k (i.e. $e_k = x(t_k) - x_k$), the following expression can be obtained:

$$e_{k+1} = e_k + h\left[f(t_k, x(t_k)) - f(t_k, x_k)\right] + hL(t_k, h)$$

If M_1 is a bound of $\frac{\partial f}{\partial x}$ in the interval, the following is an estimate of the error of each iteration in terms of the previous one:

$$|e_{k+1}| \leq (1 + hM_1)|e_k| + h|L(h)|$$

This inequality can be used to write the accumulated error as the sum of a progression where $e_0 = 0$. We denote $c = (1 + hM_1)$ and $d = h|L(h)|$, the accumulated error as $\bar{e}_n$ and then we can write:

$$|\bar{e}_n| \leq (1 + c + \ldots + c^{n-1})d = \frac{c^n - 1}{c - 1}d = \frac{(1 + hM_1)^n - 1}{hM_1}h|L(h)|$$

And thus the accumulated error is

$$|\bar{e}_n| \leq \frac{(1 + hM_1)^n}{M_1}|L(h)|$$

The numerator can be estimated remarking that it can be expressed as $(1 + x)^{\frac{k}{x}}$ where $k = (b - a)M_1$, whose limit is an exponential as x tends to zero, and thus,

$$|\bar{e}_n| \leq \frac{e^{(b-a)M_1}}{M_1}|L(h)|$$

This is an estimate of the global error of Euler's method accumulated after n iterations. As $|L(h)|$ is $0(h)$, the global error is $0(h)$ as well. Thus the solution at the end of an interval can be approximated as much as needed by making h as small as required. Because of fulfilling this property, Euler's numerical method is said to be **convergent**. In practical problems, as the global error contains an exponential, obtaining a suitable h might be difficult - perhaps it has to be so small that the number of iterations is too large.

The error of Euler's method tends to zero at the same rate as h: it is said that Euler's method is a **first order method** or that its error is first order. RK2 is **second order** as its error mimicks the behaviour of h^2, while RK4 is **fourth order**. To achieve a similar error, a first order method requires a h much smaller than a fourth order method, requiring thus a lot more iterations.

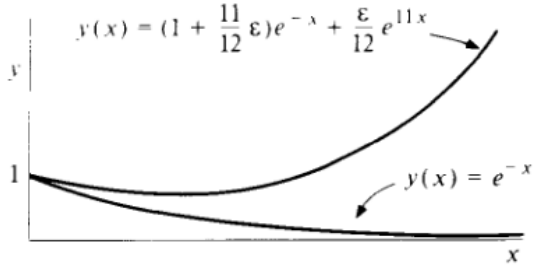
4.2.4 Stability

Stability is an important concept in DEs which impacts on their numerical solutions as well.

4.2.5 (Un)stable Solutions

It is convenient to use an example to understand the concept. Consider a second order ODE $x'' - 10x' - 11x = 0$ with initial values $x(0) = 1$, $x'(0) = -1$. It is easy to check by substitution that $x(t) = e^{-t}$ is a solution. It tends to 0 as $t \to \infty$.

Modifying the initial position by a small ϵ, i.e., $x(0) = 1 + \epsilon$, one can check that $x(t) = (1 + \frac{11}{12}\epsilon)e^{-t} + \frac{\epsilon}{12}e^{11t}$ is a solution of this slightly modified problem. The positive exponential dominates, and this solution tends to ∞ as $t \to \infty$, irrespective of how small ϵ is. The following figure represents the two solutions:



When a small change of the initial conditions leads to a very large change of the solution one speaks of an **unstable solution** or an **unstable problem**. Let us remark that this concept is independent of the numerical solution. However, if a problem is unstable, as the numerical methods always introduce small errors, it will be very difficult to solve numerically the problem as the solutions obtained can differ a lot from those intended.

4.2.6 (Un)stable Numerical Methods

Even if a DE is stable, the numerical method used to solve (approximately) the DE can be unstable.

Let us illustrate it through an example. Consider the typical first order ODE $x'(t) = f(t, x(t))$, and a two-step method $x_{k+1} = x_{k-1} + 2hf_k$, similar to Euler's. As expected from a 2-step method, its error is second order, but it is *unstable*. Consider in particular the IVP $x' = -2x + 1$, $x(0) = 1$, whose exact solution is $x(t) = \frac{1}{2}e^{-2t} + \frac{1}{2}$. This solution is stable: if the initial condition is $x(0) = 1 + \epsilon$, its exact solution is és $x(t) = (\frac{1}{2} + \epsilon)e^{-2t} + \frac{1}{2}$.

Using the exact values $x_0 = 1$ and $x_1 = \frac{1}{2}e^{-2h} + \frac{1}{2}$, $f_k = -2x_k + 1$, the 2-step numerical method applied to the IVP is expressed as:

$$x_{k+1} = x_{k-1} + 2h(-2x_k + 1) = -4hx_k + x_{k-1} + 2h$$

The absolute value of this numerical solution tends to ∞ irrespective of how small h is. The DE is stable, it is the numerical method itself which introduces the unstability.

Indeed, the numerical methods are *finite difference equations* which approximately represent DEs. The latter can be stable, but the former can be unstable, as the example shows.

Generally, implicit methods have better stability properties than explicit ones. Linear (ordinary or partial) DEs have better properties than nonlinear ones, again in general terms. Analysing these properties is complex, it has to be studied on a case by case basis. Visual simulation helps to understand what is going on.

As seen later, some numerical approximations of PDEs are stable depending on some conditions. Indeed:

- For the *explicit finite differences method for the 1D heat PDE* a stability condition is $\Delta t < \frac{(\Delta x)^2}{c(1+\cos \pi \Delta x)}$.
- For the *explicit finite differences method for the 1D wave PDE* a stability condition is $\Delta t < \frac{\Delta x}{\sqrt{c}}$.

Chapter 5

PDEs: The 1D+t Wave and Heat Equations and Numerical Solutions

5.1 Concepts and Basic Examples

5.1.1 Archetypical Examples

Consider functions such as $u(x,t)$ or $u(x,y,z,t)$, functions of two or more variables. Equations involving their partial derivatives are examples of **partial differential equations** (PDEs). The following PDEs are important examples, which are archetypical:

- The **heat equation** (in one spatial dimension) $u_t(x,t) = cu_{xx}(x,t)$ is representative of the **parabolic** PDEs.
- The **wave equation** (in one spatial dimension) $u_{tt}(x,t) = c^2u_{xx}(x,t)$ is representative of the **hyperbolic** PDEs.
- The equation $u_{xx}(x,y,z) + u_{yy}(x,y,z) + u_{zz}(x,y,z) = f(x,y,z)$ is representative of the **elliptic** PDEs. If $f = 0$ it is called the **Laplace equation**.

The t variable represents, as usual, time, while x the 1D space, the simplest case. More realistic situations consider the three spatial dimensions (x,y,z) and functions $u(x,y,z,t)$. For instance, the 3D heat equation is $u_t(x,y,z,t) = c(u_{xx}(x,y,z,t) + u_{yy}(x,y,z,t) + u_{zz}(x,y,z,t))$.

The u function represents a physical magnitude. For instance, the temperature at a point x at time t in the heat equation, where c is the thermal conductivity coefficient. The equation models the heat **diffusion** due to the temperature differences (see, for instance the wikipedia `http://en.wikipedia.org/wiki/Heat_equation`). The wave equation can be used to model the water surface waves, where $u(x,t)$ represents the level of the water surface (and c is the wave speed). We study both later in this chapter.

The parabolic, hyperbolic and elliptic PDEs have very different properties. In particular, the elliptic equation indicated above, which does not contain time, results from considering the **equilibrium solutions** of parabolic or hyperbolic PDEs.

A **solution** of a PDE is a function which satisfies the equation (and the initial and boundary conditions, as discussed next).

5.1.2 Initial and Boundary Conditions

The description of the physical system needs supplementary conditions besides the equations themselves.

As for ODEs we usually need **initial conditions**:

- The heat equation $u_t(x,t) = cu_{xx}(x,t)$ relates the speed u_t with u_{xx}; an initial condition is needed, namely, the initial distribution of temperatures $u(x,0) = u_0(x)$.
- The wave equation $u_{tt}(x,t) = c^2u_{xx}(x,t)$ relates the acceleration u_{tt} with u_{xx}; the required initial conditions are the initial positions of the surface level $u(x,0) = u_0(x)$, and the initial velocities $u_t(x,0) = v_0(x)$.

Let us *remark* that elliptic PDEs do not contain time, and *do not* require initial conditions. *Remark* as well that the initial conditions are functions of x (not just values, as for ODEs).

PDEs usually require **boundary conditions** as well to complete the description of the physical system. For instance, if we are simulating the surface waves, the walls of the container would be the boundary. The boundary conditions specify what happens at the (spatial) boundary along the time. In the case of the heat equation, the body considered might be:

- *thermically isolated*, which is represented by establishing that the normal derivative at the boundary is zero, i.e., $\frac{\partial u}{\partial n}(x,t) = 0$, $\forall x \in Boundary$, $\forall t \geq 0$: it is said a homogeneous **Neumann boundary condition**;
- in an *environment at a fixed temperature* $u(x,t) = u_B$, $\forall x \in Boundary$, $\forall t \geq 0$: it is said a **Dirichlet boundary condition**;
- the **mixed boundary conditions** combine $\frac{\partial u}{\partial n}$ and u.

In the case of the surface waves in a container, the most natural boundary condition is Neumann homogeneous: the normal derivative 0 models that the waves are reflected at the walls.

5.2 Finite Differences Discretization

To be able to compute numerical solutions of the above described PDEs, we must find a way to express each partial derivative (u_x, u_t, u_{tt}, u_{xx}) in term of discretized sections of the systems. For this purpose, finite differences of the partial derivatives will be applied to obtain expressions that we can use to update our system. The objective is to find expresssions that relate the next timestep (u^{m+1}), which is unknown, to the current (u^m) and previous timesteps (u^{m-1}), allowing to write the partial differential equation as a function of discretized sections of the physical system to be simulated.

To write these discretizations, the notation required is a bit more complex than for ODEs. Consider a time instant $m\Delta t$ and a spatial position $j\Delta x$ (we assume $N\Delta x = l$; we denote by u_j^m the approximate value at those instant and position, i.e.:

$$u_j^m \simeq u(j\Delta x, m\Delta t), \forall m, \quad \text{with } 0 \leqslant j \leqslant N$$

One can discretize the left hand side (LHS) and right hand side (RHS) of a PDE by finding the finite difference discretization of all contained partial derivatives. This is done in a similar way as in Euler's method; take as an example the $u_t(x,y)$ term, which is a first order partial derivative with respect to the time:

$$\frac{\partial u(j\Delta x, m\Delta t)}{\partial t} \simeq \frac{u(j\Delta x, (m+1)\Delta t) - u(j\Delta x, m\Delta t)}{\Delta t} \simeq \frac{u_j^{m+1} - u_j^m}{\Delta t}$$

whose local discretization error is first order.

Similarly, second order partial derivatives (such as $u_{xx}(x,t)$ or $u_{tt}(x,t)$) are discretized following expressions in this way:

$$\frac{\partial^2 u(j\Delta x, m\Delta t)}{\partial x^2} \simeq \frac{u_{j+1}^m - 2u_j^m + u_{j-1}^m}{\Delta x^2}$$

whose local discretization error is second order. Similarly, the (explicit) temporal discretization of a second order temporal derivative is the following:

$$\frac{\partial^2 u(j\Delta x, m\Delta t)}{\partial t^2} \simeq \frac{u_j^{m+1} - 2u_j^m + u_j^{m-1}}{\Delta t^2}$$

Many finite differences exist: we just applied the explicit finite differences, but the implicit and Crank-Nicolson finite differences also exist. The implicit discretization is characterized by taking information of the next timestep (t_{m+1}) in the spatial derivatives, obtaining a similar expression at the $m+1$ timestamp:

$$\frac{\partial^2 u(j\Delta x, m\Delta t)}{\partial x^2} \simeq \frac{u_{j+1}^{m+1} - 2u_j^{m+1} + u_{j-1}^{m+1}}{\Delta x^2},$$

whereas the Crank-Nicolson discretization, on its behalf, consists in an averaging of the explicit and implicit finite differences expressions for the spatial derivatives:

$$\frac{\partial^2 u(j\Delta x, m\Delta t)}{\partial x^2} \simeq \frac{1}{2}\frac{u_{j+1}^m - 2u_j^m + u_{j-1}^m}{\Delta x^2} + \frac{1}{2}\frac{u_{j+1}^{m+1} - 2u_j^{m+1} + u_{j-1}^{m+1}}{\Delta x^2}$$

In the following sections we will see how to handle the expressions resulting from substituting the discretizations into the different PDEs (the heat and wave equations). Take into account that, at any point in our simulation, the values of u^{m-1} and u^m will be known, and as we are attempting to compute the value of the next timestep (u^{m+1}).

For explicit formulations the solution is obvious, as there is only one u^{m+1} term (m represents time) and the same techniques we have been applying in the previous chapters could be applied; however, when using implicit or Crank-Nicolson approximations, many u^{m+1} terms will appear in the expression, difficulting their computation.

For those cases the numerical solution will be obtained by (iteratively) computing the approximation at an instant from the previous approximations using different solvers, as we will see in the following sections. For the application of these solvers, we need to isolate all u^{m+1} terms at the RHS of the equation for posteriorly forming a linear system of equations. That is, for our explicit example:

$$u_j^{m+1} = u_j^m + \frac{c\Delta t}{\Delta x^2}(u_{j+1}^m - 2u_j^m + u_{j-1}^m),$$

although this will vary depending on the discretization, as we will see in the following sections.

5.2.1 Numerical Solution of the 1D Heat Equation: Finite Differences, Explicit Method

We start with the heat equation, which is slightly easier than the wave equation. The 1D heat equation is:

$$u_t(x,t) = cu_{xx}(x,t) \qquad x \in (0,l) \qquad t \in \mathbb{R}^+$$

supplemented with *initial conditions* $u(x,0) = u_0(x), x \in (0,l)$ and *boundary conditions*, for instance homogeneous Dirichlet, $u(0,t) = 0 = u(l,t), \forall t \in \mathbb{R}^+$.

Both sides of the equation contain (partial) derivatives, i.e., limits. To be able to compute its numerical solution, we approximate each partial derivative term by a fraction, as in Euler's method. This is called the **discretization** of a partial derivative. To write its discretization, the notation required is a bit more complex than for ODEs. Consider a time instant $m\Delta t$ and a spatial position $j\Delta x$ (we assume $N\Delta x = l$; we denote by u_j^m the approximate value at those instant and position, i.e.:

$$u_j^m \simeq u(j\Delta x, m\Delta t), \forall m, 0 \leqslant j \leqslant N$$

One can discretize the left hand side (LHS) and right hand side (RHS) of a PDE by finding the finite difference discretization of all contained terms. Take as an example the heat equation: $u_t(x,t) = cu_{xx}(x,t)$; if we found the discretization of both $u_t(x,t)$ and $u_{xx}(x,t)$ terms, we could find an expression for updating the value of out numerical simulation. This is done in a similar way as in Euler's method; take as an example the $u_t(x,y)$ term:

$$\frac{\partial u(j\Delta x, m\Delta t)}{\partial t} \simeq \frac{u(j\Delta x, (m+1)\Delta t) - u(j\Delta x, m\Delta t)}{\Delta t} \simeq \frac{u_j^{m+1} - u_j^m}{\Delta t}$$

whose local discretization error is first order.

The right hand side (RHS) of our example, the heat equation, contains $u_{xx}(x,t)$, which is a kind of spatial acceleration, and it is usually discretized as:

$$\frac{\partial^2 u(j\Delta x, m\Delta t)}{\partial x^2} \simeq \frac{u_{j+1}^m - 2u_j^m + u_{j-1}^m}{\Delta x^2}$$

whose local discretization error is second order.

Equating both approximations, the discretized 1D heat equation is:

$$\frac{1}{\Delta t}(u_j^{m+1} - u_j^m) = \frac{c}{\Delta x^2}(u_{j+1}^m - 2u_j^m + u_{j-1}^m)$$

This type of approximation is known as **finite differences**. Many types of finite difference discretizations exist: the one we just applied is the explicit finite differences, but throughout the course we will also see implicit and Crank-Nicolson finite differences. The implicit discretization is characterized by being computed at the next timestep (t_{m+1}), obtaining the expression at the $m+1$ timestamp:

$$\frac{\partial^2 u(j\Delta x, m\Delta t)}{\partial x^2} \simeq \frac{u_{j+1}^{m+1} - 2u_j^{m+1} + u_{j-1}^{m+1}}{\Delta x^2},$$

whereas the Crank-Nicolson discretization consists in an averaging of the explicit and implicit finite differences expressions:

$$\frac{\partial^2 u(j\Delta x, m\Delta t)}{\partial x^2} \simeq \frac{1}{2}\frac{u_{j+1}^{m} - 2u_j^{m} + u_{j-1}^{m}}{\Delta x^2} + \frac{1}{2}\frac{u_{j+1}^{m+1} - 2u_j^{m+1} + u_{j-1}^{m+1}}{\Delta x^2}$$

Finally, the (explicit) temporal discretization of a second order temporal derivative is the following:

$$\frac{\partial^2 u(j\Delta x, m\Delta t)}{\partial x^2} \simeq \frac{u_{j+1}^{m+1} - 2u_j^{m+1} + u_{j-1}^{m+1}}{\Delta x^2},$$

In the following sections we will see how to handle the expressions resulting from substituting the discretizations into the different PDEs. For explicit formulations the solution is obvious, as there is only one u^{m+1} term (m represents time) and the same techniques we have been applying in the previous chapters could be applied; however, when using implicit or Crank-Nicolson approximations, many u^{m+1} terms will appear in the expression, difficulting their computation.

For those cases the numerical solution will be obtained by (iteratively) computing the approximation at an instant from the previous approximations using different solvers, as we will see in the following sections. For the application of these solvers, we need to isolate all u^{m+1} terms at the RHS of the equation for posteriorly forming a linear system of equations. That is, for our explicit example:

$$u_j^{m+1} = u_j^m + \frac{c\Delta t}{\Delta x^2}(u_{j+1}^m - 2u_j^m + u_{j-1}^m),$$

although this will vary depending on the discretization, as we will see in the following sections.

Remark: for forming the linear system of equations we have to take into account that the above equation is repeated as many times as there are js (more precisely, for $1 \leqslant j \leqslant N-1$, as the boundary conditions hold for $j = 0$ and $j = N$): programming-wise this is simple by using arrays. Each instant needs an array and not just a value, as for ODEs.

Using $\mu = \frac{c\Delta t}{\Delta x^2}$ the notation becomes simpler:

$$u_j^{m+1} = u_j^m + \mu(u_{j+1}^m - 2u_j^m + u_{j-1}^m), \text{ with } 1 \leqslant j \leqslant N-1$$

This is the practical expression to be used for the computation of the numerical solution, adding the initial condition, $u_j^0 = u(j\Delta x, 0)$ $0 \leqslant j \leqslant N$, to initialize the iterations.

This finite difference method is **explicit**. Explicit methods can lead to (numerical) instabilities. It can be shown that a necessary condition for this method to be stable is:

$$\Delta t < \frac{(\Delta x)^2}{c(1+\cos \pi \Delta x)}$$

In summary, the method is:

$$u_j^0 = u(j\Delta x, 0), 0 \leqslant j \leqslant N$$

$$\forall m \geqslant 0, u_j^{m+1} = u_j^m + \mu(u_{j+1}^m - 2u_j^m + u_{j-1}^m), 1 \leqslant j \leqslant N-1$$

where $\mu = \frac{c\Delta t}{\Delta x^2}$, plus the corresponding *boundary conditions*.

Exercise 18

Use the previous finite difference scheme to solve the 1D heat equation $u_t(x,t) = u_{xx}(x,t)$ in the $x \in (0,\pi)$ interval for the initial distribution of temperatures $u_0(x) = \sin x$ with (homogeneous Dirichlet) boundary conditions $u(0,t) = 0 = u(\pi,t)$. Δx and Δt have to be chosen. $\Delta x = \pi/16$ can be a convenient choice. For Δt 0.1 or 0.01 could be chosen. Compare the solution in both cases.

5.2.2 Numerical Solution of the 1D Wave Equation: Finite Differences, Explicit Method

The 1D wave equation can be used to simulate the water surface waves. The function will be denoted by h instead of u, because the magnitude is the height, the vertical level of the water surface. The 1D wave equation we use is $h_{tt}(x,t) = c^2 h_{xx}(x,t)$; as in the previous subsection $h_j^m \simeq h(j\Delta x, m\Delta t)$ is the approximate solution. In both sides of the equation there are two "accelerations", which we discretize as in the previous subsection as:

$$\frac{\partial^2 h(j\Delta x, m\Delta t)}{\partial x^2} \simeq \frac{h_{j+1}^m - 2h_j^m + h_{j-1}^m}{\Delta x^2}$$

and

$$\frac{\partial^2 h(j\Delta x, m\Delta t)}{\partial t^2} \simeq \frac{h_j^{m+1} - 2h_j^m + h_j^{m-1}}{\Delta t^2}$$

Equating both, the discretized 1D wave equation is:

$$\frac{h_j^{m+1} - 2h_j^m + h_j^{m-1}}{\Delta t^2} = c^2 \frac{h_{j+1}^m - 2h_j^m + h_{j-1}^m}{\Delta x^2}$$

Simplifying notation using:

$$\mu = c^2 \frac{\Delta t^2}{\Delta x^2}$$

we obtain:

$$h_j^{m+1} - 2h_j^m + h_j^{m-1} = \mu(h_{j+1}^m - 2h_j^m + h_{j-1}^m)$$

Now we should isolate h_j^{m+1}, and we have:

$$h_j^{m+1} = 2h_j^m - h_j^{m-1} + \mu(h_{j+1}^m - 2h_j^m + h_{j-1}^m), 1 \leqslant j \leqslant N-1$$

which can be used to compute the instantaneous height from previously computed ones. It is an **explicit** formula, where arrays must be used for easier programming. The initial position $h_0(x)$ provides the values $h_j^0 = h_0(j\Delta x)$. Using $h_0(x)$, $v_0(x)$ and RK2 in time h_j^1 can be obtained. These values are needed to initialize the iterative loop. *Remark*: never forget the *boundary conditions* prescribed in the problem for $j = 0$ and $j = N$.

A sufficient condition of stability of this method for the wave equation is $\Delta t < \frac{\Delta x}{\sqrt{c}}$.

Exercise 19
Use the previous finite difference scheme to solve the 1D wave equation $h_{tt}(x,t) = h_{xx}(x,t)$ in the $x \in (0,\pi)$ interval for the initial distribution of heights $h_0(x) = 1 + 0.1\sin x$, zero initial speed, with boundary conditions $h(0,t) = 1 = h(\pi,t)$. Use Δx and Δt as in the previous exercise.

Exercise 20
Reformulate the previous two exercises using homogeneous Neumann boundary conditions, i.e., $u_x(0,t) = 0 = u_x(\pi,t)$ for the heat equation and $h_x(0,t) = 0 = h_x(\pi,t)$ for the wave equation. Compare the solutions of this exercise with those of the previous ones.

5.2.3 Implicit Finite Differences for the 1D+t Wave Equation

The previous explicit method can be transformed into an **implicit** one equating the discretization of the second temporal derivative at m, j with the discretization of the second spatial derivative at $m+1, j$, i.e.,:

$$\frac{h_j^{m+1} - 2h_j^m + h_j^{m-1}}{\Delta t^2} = c^2 \frac{h_{j+1}^{m+1} - 2h_j^{m+1} + h_{j-1}^{m+1}}{\Delta x^2}$$

and using $\mu = c^2 \frac{\Delta t^2}{\Delta x^2}$ to have a lighter notation one gets:

$$h_j^{m+1} - 2h_j^m + h_j^{m-1} = \mu(h_{j+1}^{m+1} - 2h_j^{m+1} + h_{j-1}^{m+1})$$

As usual, we group the $m+1$ terms (the next iteration) on the LHS:

$$-\mu h_{j+1}^{m+1} + (1+2\mu)h_j^{m+1} - \mu h_{j-1}^{m+1} = 2h_j^m - h_j^{m-1}, 1 \leqslant j \leqslant N-1$$

which gives us the implicit finite differences expression for the 1D wave equation, the iteration formula. Let us *remark* that this formula expresses in an extremely compressed way a **linear system of many differential equations and unknowns**: the unknows are h_j^{m+1}, for $1 \leqslant j \leqslant N-1$. There are as many equations as unknowns. From previous iterations, the h_j^m and h_j^{m-1} appearing on the RHS are known - to start the iterations, let us recall that we need two initial conditions h_j^1 and h_j^0 -.

Let us write more explicitly this linear system in a matrix form $Ah = b$ where:

- h is the column vector of the unknowns h_j^{m+1}
- A is the coefficients matrix; most of them are zero, except for $-\mu, (1+2\mu), -\mu$, which appear on the diagonal and close to it. It is said that this matrix is **tridiagonal**
- b is the column vector of the independent terms, those on the RHS.

Thus (writing for A only the non-zero coefficients):

$$A = \begin{bmatrix} 1+2\mu & -\mu & & & & & \cdots \\ -\mu & 1+2\mu & -\mu & & & & \cdots \\ & -\mu & 1+2\mu & -\mu & & & \cdots \\ \vdots & & \ddots & \ddots & \ddots & & \cdots \\ \cdots & & & -\mu & 1+2\mu & -\mu & \\ \cdots & & & & -\mu & 1+2\mu & -\mu \\ \cdots & & & & & -\mu & 1+2\mu \end{bmatrix}$$

and the column vectors are:

$$h = \begin{bmatrix} h_1^{m+1} \\ h_2^{m+1} \\ h_3^{m+1} \\ \vdots \\ h_{N-3}^{m+1} \\ h_{N-2}^{m+1} \\ h_{N-1}^{m+1} \end{bmatrix} \qquad b = \begin{bmatrix} 2h_1^m - h_1^{m-1} \\ 2h_2^m - h_2^{m-1} \\ 2h_3^m - h_3^{m-1} \\ \vdots \\ 2h_{N-4}^m - h_{N-3}^{m-1} \\ 2h_{N-3}^m - h_{N-2}^{m-1} \\ 2h_{N-2}^m - h_{N-1}^{m-1} \end{bmatrix}$$

To be more precise, let us *remark* that the values h_0^m and h_N^m are at the *boundary*. They are determined by the boundary conditions, thus they are not in the linear system.

- Indeed, if the boundary conditions are **Dirichlet**, the values h_0^m and h_N^m are those given by the conditions. The first and last equations have to be modified, these known values are included in the RHS, namely, the *first and last rows of the column vector* b will contain $+\mu h_0$ and $+\mu h_N$, respectively. For instance, the first equation $-\mu h_0^{m+1} + (1+2\mu)h_1^{m+1} - \mu h_2^{m+1} = 2h_1^m - h_1^{m-1}$ becomes $(1+2\mu)h_1^{m+1} - \mu h_2^{m+1} = 2h_1^m - h_1^{m-1} + \mu h_0$. In fact, the linear system $Ah = b$ shown above corresponds to homogeneous Dirichlet boundary conditions.
- If the boundary conditions are **Neumann homogeneous**, a way to impose these conditions is that the values at the boundary are equal to those immediately next, e.g., $h_0^m = h_1^m$ and $h_N^m = h_{N-1}^m$. The *first and last equations of the system are modified*, $1+2\mu$ *is replaced by* $1+\mu$. For instance, in the RHS of the first equation we have $-\mu h_0^{m+1} + (1+2\mu)h_1^{m+1} - \mu h_2^{m+1} = -\mu h_1^{m+1} + (1+2\mu)h_1^{m+1} - \mu h_2^{m+1} = (1+\mu)h_1^{m+1} - \mu h_2^{m+1}$, and thus the equation becomes $(1+\mu)h_1^{m+1} - \mu h_2^{m+1} = 2h_1^m - h_1^{m-1}$. If there are few spatial points, an alternative way of applying the boundary conditions, which diminishes the error, is to add *extra points* outside the domain, h_{-1} and h_{N+1}, and impose $h_0 = h_{-1}$ and $h_{N+1} = h_N$.

5.2.4 Implicit Finite Differences for the 1D+t Heat Equation

In a similar way, the implicit formulation for the heat equation results from equating the discretization of the first time derivative at m, j to the one of the second spatial derivative at $m+1, j$, namely:

$$\frac{1}{\Delta t}(u_j^{m+1} - u_j^m) = \frac{c}{\Delta x^2}(u_{j+1}^{m+1} - 2u_j^{m+1} + u_{j-1}^{m+1})$$

Denoting $\mu = c\frac{\Delta t}{\Delta x^2}$ we get

$$u_j^{m+1} - u_j^m = \mu(u_{j+1}^{m+1} - 2u_j^{m+1} + u_{j-1}^{m+1})$$

Putting the $m+1$ terms on the LHS and moving those previously computed to the RHS:

$$-\mu u_{j+1}^{m+1} + (1+2\mu)u_j^{m+1} - \mu u_{j-1}^{m+1} = u_j^m, 1 \leqslant j \leqslant N-1$$

We have again a **(large) system of linear equations** to be solved at each iteration. Its matrix expression is $Au = b$ with u the column vector of unknowns u_j^{m+1}, A the tridiagonal matrix of the coefficients, and b the column vector of independent terms, as follows:

$$A = \begin{bmatrix} 1+2\mu & -\mu & & & & & \cdots \\ -\mu & 1+2\mu & -\mu & & & & \cdots \\ & -\mu & 1+2\mu & -\mu & & & \cdots \\ \vdots & & \ddots & \ddots & \ddots & & \cdots \\ \cdots & & & -\mu & 1+2\mu & -\mu & \\ \cdots & & & & -\mu & 1+2\mu & -\mu \\ \cdots & & & & & -\mu & 1+2\mu \end{bmatrix}$$

and vectors

$$u = \begin{bmatrix} u_1^{m+1} \\ u_2^{m+1} \\ u_3^{m+1} \\ \vdots \\ u_{N-3}^{m+1} \\ u_{N-2}^{m+1} \\ u_{N-1}^{m+1} \end{bmatrix} \qquad b = \begin{bmatrix} u_1^m \\ u_2^m \\ u_3^m \\ \vdots \\ u_{N-3}^m \\ u_{N-2}^m \\ u_{N-1}^m \end{bmatrix}$$

We only wrote for A the non-zero coefficients. The boundary conditions, **Dirichlet** or **Neumann**, are treated in a similar way as in the wave equation. The matrix A and vector b written above correspond to homogeneous Dirichlet boundary conditions.

The linear systems are nonsingular. They can be solved through LU, but when they are very large it is more convenient to use *iterative methods*, such as Jacobi or Gauss-Seidel. We discuss them later.

Jacobi's method comes from isolating the diagonal terms and computing them iteratively:

$$u_j^{m+1,(n+1)} = \frac{u_j^m}{(1+2\mu)} + \frac{\mu}{(1+2\mu)}(u_{j+1}^{m+1,(n)} + u_{j-1}^{m+1,(n)})$$

Gauss-Seidel's method is very similar:

$$u_j^{m+1,(n+1)} = \frac{u_j^m}{(1+2\mu)} + \frac{\mu}{(1+2\mu)}(u_{j+1}^{m+1,(n)} + u_{j-1}^{m+1,(n+1)})$$

5.2.5 Crank-Nicolson's method

The Crank-Nicolson's implicit method is very frequently used to solve numerically the wave and heat equations. It is derived from averaging the explicit and implicit discretizations of the spatial acceleration:

$$\frac{\partial^2 h(j\Delta x, m\Delta t)}{\partial x^2} \simeq \frac{1}{2}\frac{h_{j+1}^m - 2h_j^m + h_{j-1}^m}{\Delta x^2} + \frac{1}{2}\frac{h_{j+1}^{m+1} - 2h_j^{m+1} + h_{j-1}^{m+1}}{\Delta x^2}$$

1D+t Wave Equation

Indeed, for the wave equation we have:

$$\frac{h_j^{m+1} - 2h_j^m + h_j^{m-1}}{\Delta t^2} = c^2\left(\frac{1}{2}\frac{h_{j+1}^m - 2h_j^m + h_{j-1}^m}{\Delta x^2} + \frac{1}{2}\frac{h_{j+1}^{m+1} - 2h_j^{m+1} + h_{j-1}^{m+1}}{\Delta x^2}\right)$$

Denoting $\mu = c^2 \frac{\Delta t^2}{2\Delta x^2}$ (*remark* the 2 added in the denominator):

$$h_j^{m+1} - 2h_j^m + h_j^{m-1} = \mu[(h_{j+1}^m - 2h_j^m + h_{j-1}^m) + (h_{j+1}^{m+1} - 2h_j^{m+1} + h_{j-1}^{m+1})]$$

Moving the suitable terms to the RHS and LHS:

$$-\mu h_{j+1}^{m+1} + (1+2\mu)h_j^{m+1} - \mu h_{j-1}^{m+1} = \mu h_{j+1}^m + 2(1-\mu)h_j^m + \mu h_{j-1}^m - h_j^{m-1}$$

i.e., it results a system of linear equations $Ah = b$ to be solved at each iteration. Comparing with previous systems, the matrix A has the same expression, although the *value of* μ *is different*, the unknowns vector h is the same, and the independent terms column vector b is quite different. Boundary conditions are dealt with as previously.

The corresponding Jacobi and Gauss-Seidel formulations are:

$$h_j^{m+1,(n+1)} = \frac{\mu h_{j+1}^m + 2(1-\mu)h_j^m + \mu h_{j-1}^m - h_j^{m-1}}{(1+2\mu)} + \frac{\mu}{(1+2\mu)}(h_{j+1}^{m+1,(n)} + h_{j-1}^{m+1,(n)})$$

$$h_j^{m+1,(n+1)} = \frac{\mu h_{j+1}^m + 2(1-\mu)h_j^m + \mu h_{j-1}^m - h_j^{m-1}}{(1+2\mu)} + \frac{\mu}{(1+2\mu)}(h_{j+1}^{m+1,(n)} + h_{j-1}^{m+1,(n+1)})$$

1D+t Heat Equation

For the heat equation we have:

$$\frac{u_j^{m+1} - u_j^m}{\Delta t} = c\left(\frac{1}{2}\frac{u_{j+1}^m - 2u_j^m + u_{j-1}^m}{\Delta x^2} + \frac{1}{2}\frac{u_{j+1}^{m+1} - 2u_j^{m+1} + u_{j-1}^{m+1}}{\Delta x^2}\right)$$

Now $\mu = c\frac{\Delta t}{2\Delta x^2}$ gives the simpler:

$$u_j^{m+1} - u_j^m = \mu[(u_{j+1}^m - 2u_j^m + u_{j-1}^m) + (u_{j+1}^{m+1} - 2u_j^{m+1} + u_{j-1}^{m+1})]$$

Grouping suitably the terms on the RHS and LHS:

$$-\mu u_{j+1}^{m+1} + (1+2\mu)u_j^{m+1} - \mu u_{j-1}^{m+1} = \mu u_{j+1}^m + (1-2\mu)u_j^m + \mu u_{j-1}^m$$

leading to a system of linear equations $Au = b$, which depends on the boundary conditions.

The corresponding Jacobi and Gauss-Seidel formulations are:

$$u_j^{m+1,(n+1)} = \frac{\mu u_{j+1}^m + (1-2\mu)u_j^m + \mu u_{j-1}^m}{(1+2\mu)} + \frac{\mu}{(1+2\mu)}(u_{j+1}^{m+1,(n)} + u_{j-1}^{m+1,(n)})$$

$$u_j^{m+1,(n+1)} = \frac{\mu u_{j+1}^m + (1-2\mu)u_j^m + \mu u_{j-1}^m}{(1+2\mu)} + \frac{\mu}{(1+2\mu)}(u_{j+1}^{m+1,(n)} + u_{j-1}^{m+1,(n+1)})$$

Exercise 21

- *Use implicit and Crank-Nicolson to solve the 1D+t heat equation* $u_t(x,t) = u_{xx}(x,t)$ *in the* $x \in (0,\pi)$ *interval for the initial distribution of temperatures* $u_0(x) = \sin x$*, both with homogeneous Dirichlet and homogeneus Neumann boundary conditions (*$u(0,t) = 0 = u(\pi,t)$ *and* $u_x(0,t) = 0 = u_x(\pi,t)$ *respectively) using* $\Delta x = \pi/16$ *and* $\Delta t = 0.1$*. Compare the solutions among them, and with those obtained using explicit finite differences.*

- *Use implicit and Crank-Nicolson to solve the 1D+t wave equation* $h_{tt}(x,t) = h_{xx}(x,t)$ *in the* $x \in (0,\pi)$ *interval for the initial distribution of heights* $h_0(x) = 1 + 0.1\sin x$*, zero initial speed, with Dirichlet boundary conditions* $h(0,t) = 1 = h(\pi,t)$ *and with homogeneus Neumann boundary conditions, using* $\Delta x = \pi/16$ *and* $\Delta t = 0.1$*. Compare the solutions among them, and with those obtained using explicit finite differences.*

- *Formulate precisely the systems of linear equations for the wave and heat equations resulting from Crank-Nicolson according to different boundary conditions, namely, Dirichlet and Neumann homogeneous.*

5.3 Iterative Numerical Solution of Linear Systems

5.3.1 Motivation of Using Iterative Solving of Linear Systems

Systems of linear equations can be efficiently solved by **Gaussian elimination** or, more precisely, by **LU**. However, when the dimension of the systems is very large, as it happens in the systems obtained in implicit methods for PDEs, **iterative techniques** can be more appropriate. Let us briefly discuss why.

Consider a simulation of the water waves on the sea surface based on the 2D +t wave equation, which needs to be visualised on a screen; assume, conservatively, that the screen is 1024x1024 pixels. One would need an h per pixel, and thus the number of unknowns would approximately be 10^6, and the matrix of the linear system to be solved at each iteration would have a 10^6x$10^6 = 10^{12}$ size. The storage for this matrix, if each element is a *float*, represented by a byte, would be of the order of Terabytes, and more would be needed for temporary computations. Although in this and other cases the matrices are mostly made of zeros (called *sparse matrices*), using LU requires a huge number of computations before the solution is obtained, despite we need real time simulations; for these, and for symmetry reasons LU is not suitable, and we should consider iterative strategies as more suitable than direct ones. Thus we turn to iterative methods.

5.3.2 Jacobi's Method

Jacobi's Algorithm

Jacobi's method is the most elementary iterative method to solve systems of linear equations. Suppose that the system to be solved is $Ax = b$, where x and b are N dimensional column vectors and A an NxN matrix. We require the diagonal terms to be non zero, i.e., $a_{ii} \neq 0$, $i = 1, 2, \ldots, N$. Starting from any N-dimensional initial vector $x^{(0)}$, Jacobi's algorithm iteratively gives the different components of the next vector as:

$$x_i^{(n+1)} = \frac{b_i - \sum_{j \neq i} a_{ij} x_j^{(n)}}{a_{ii}}$$

((informally speaking, this algorithm comes from isolating the x_i at the diagonal) Under certain conditions these iterations reach a limit, which is **the solution of the system of linear equations**. We discuss why next.

Jacobi's Method as a Fixed Point Case

Let us consider a transformation f. We iteratively define $x^{(n+1)} = f(x^{(n)})$. If $\lim_{n\to\infty} x^{(n)}$ exists (denoted as x^*), and if $x^* = f(\lim_{n\to\infty} x^{(n)})$ - this happens if f is continuous, for instance -, then $f(x^*) = x^*$, and we say that x^* is a **fixed point of the transformation** f.

Jacobi's method is an example of fixed point: the algorithm can be written in matrix form as:

$$x^{(n+1)} = D^{-1}(b + \tilde{A}x^{(n)})$$

where

$$D = diag(a_{11}, \ldots, a_{NN}) \qquad \tilde{A} = -\begin{bmatrix} 0 & a_{12} & . & . & a_{1N} \\ a_{21} & 0 & . & . & . \\ . & . & . & . & . \\ . & . & . & . & . \\ a_{N1} & . & . & . & 0 \end{bmatrix}$$

If $\lim_{n\to\infty} x^{(n+1)} = x^*$ exists, then x^* is a fixed point, as linear transformations (represented by matrices) are continuous. Let us see that this limit, if it exists, is the solution of the system of linear equations. If the limit exists, then $\lim_{n\to\infty} x^{(n)} = \lim_{n\to\infty} x^{(n+1)}$, and we take limits in Jacobi's algorithm getting:

$$\lim_{n\to\infty} x^{(n+1)} = \lim_{n\to\infty} \left[D^{-1}(b + \tilde{A}x^{(n)})\right]$$

and thus:

$$x^* = D^{-1}(b + \tilde{A}x^*)$$

Multiplying both sides by D we obtain $Dx^* = b + \tilde{A}x^*$ and $(D - \tilde{A})x^* = b$. Finally, $Ax^* = b$: as this limit satisfies the system of linear equations in matrix form, it is its solution.

Convergence of Jacobi's Method

Jacobi's method is said **convergent** if the iteration has a limit - which is the system solution -. In Jacobi's algorithm we can make $y = D^{-1}(b)$ and $D^{-1}\tilde{A} = T$, and the iteration can be more concisely expressed as $x^{(n+1)} = y + Tx^{(n)} (= f(x^{(n)}))$. In the general case of fixed point of a transformation f, it can be shown that if the absolute value of the *derivative* of the transformation is less than 1, the iteration has a unique limit, the fixed point, which can be obtained for any initial value $x^{(0)}$. Let us remark that we would need to define the *derivative* of a transformation.

In Jacobi's method, the derivative of the transformation $x \to y + Tx$ is T, because of linearity. If the *spectral radius* of T (the maximum of the absolute value of the eigenvalues of T) is less than 1, the absolute value of the *derivative* is less than 1, and Jacobi's method is convergent. A necessary condition for the spectral radius of $T = D^{-1}\tilde{A}$ to be less than 1 is that the matrix A is *diagonally dominant*, i.e., for each row i, the absolute value of the diagonal term exceeds the sum of the absolute values of the off-diagonal terms $|a_{ii}| > \sum_{j \neq i} |a_{ij}|$.

The matrices A appearing in the implicit methods to solve the PDEs discussed above are diagonally dominant as $1 + 2\mu > 2\mu$. Thus, when solving the systems of linear equations via Jacobi, we have convergent iterations.

Stopping Criteria

Jacobi's method $x^{(n+1)} = D^{-1}(b + \tilde{A}x^{(n)})$ can go on indefinitely. We need to stop at some stage. A stopping criterion is that

$$\epsilon = \frac{||x^{(k+1)} - x^{(k)}||}{||x^{(k+1)}||}$$

is smaller than a certain amount - this amount becomes an error estimate -; the numerator of the fraction is inspired in the Cauchy condition for the limit of a sequence. Another stopping criterion is to stop when the number of iterations reaches a certain k; it is a pragmatic one. In the latter case, there might be a convergence issue.

5.3.3 Gauss-Seidel's Method

This method is a small variant of Jacobi: from any initial N-dimensional vector $x^{(0)}$, iteratively compute :

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

where the updated components are used as soon as they become ready. Indeed, when we compute $x_i^{(n+1)}$, the values $x_j^{(n+1)}$ for $j < i$ have been already computed, while for $j > i$ it is not the case, so that $x_j^{(n)}$ must be used.

The matrix expression of Gauss-Seidel results from the decomposition of the matrix A as $A = D - U - L$, where U and L are upper and lower triangular, respectively:

$$U = -\begin{bmatrix} 0 & a_{12} & . & . & a_{1N} \\ 0 & 0 & . & . & . \\ . & . & . & . & . \\ . & . & . & . & a_{N-1,N} \\ 0 & . & . & . & 0 \end{bmatrix} \qquad L = -\begin{bmatrix} 0 & 0 & . & . & 0 \\ a_{21} & 0 & . & . & . \\ . & . & . & . & . \\ . & . & . & . & 0 \\ a_{N1} & a_{N2} & . & . & 0 \end{bmatrix}$$

so that $x^{(n+1)} = (D - L)^{-1}(b + Ux^{(n)})$. If x^* were a fixed point, then $x^* = (D - L)^{-1}(b + Ux^*)$ and thus, $(D - L)x^* = b + Ux^*$, and finally $(D - L - U)x^* = Ax^* = b$.

Exercise 22
Compute the first 3 iterations of the solution of the following system using Jacobi's method:

$3x - y + z = -1$

$x + 4y - z = 6$

$x - y + 3z = -3$

Estimate the absolute and relative errors after the third iteration.

EXERCISE 23
Compute the first 3 iterations of the solution of the following system using both Jacobi's and Gauss-Seidel's methods, and compare them:

$$4x - y + z = -1$$

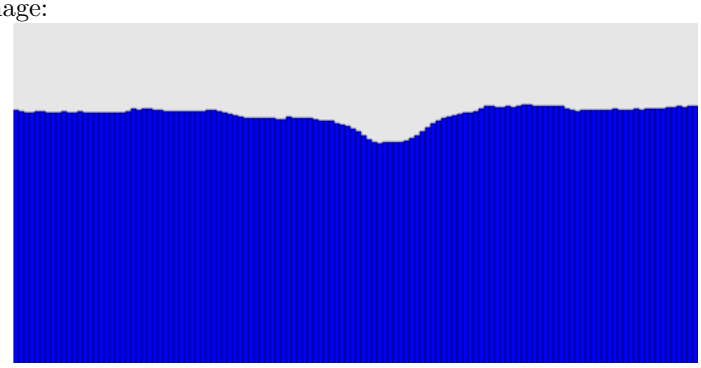
$$x + 3y - z = 6$$

$$x - y + 4z = -3$$

Estimate the absolute and relative errors after the third iteration. Formulate stopping criteria.

5.4 Re-using the Spring Numerical Solutions for the Numerical Solutions and Simulations of the 1D Wave Equation

The 1D wave equation can be used to simulate the water surface waves, as represented in the following image:



The LHS of the elastic spring equation is a (temporal) acceleration, the RHS is a function of x. A first numerical method to solve the 1D wave equation is based on re-using the approximations for the elastic spring equation. For instance, using RK4 in time and replacing the $-\frac{\mathbf{k}}{\mathbf{m}}x$ appearing on the RHS of the elastic spring by the discretization of the RHS of the wave equation h_{xx} through the usage of finite differences method:

$$\frac{\partial^2 h}{\partial x^2} \simeq \frac{h(x+\Delta x) - 2h(x) + h(x-\Delta x)}{\Delta x^2}$$

This approximation of the second derivative would be coded as:

```
def approximate_uxx(array):
    uxx = [0.0]*ArraySize
    for i in range(1, ArraySize-1):
        uxx[i] = (array[i+1] - 2*array[i] + array[i-1])/sq(delta_x)
    return uxx
```

Let us remark that

- we use **arrays** to compute the heights at each time step, instead of just one position per time step in the elastic spring
- the loop is not applied for the first and last elements of the array, whose values are determined by the **boundary conditions** at a later stage.

5.4.1 Computing with Arrays

The State

The definition of the state for the 1D wave equation could be coded as:

```
ArraySize = 64
State = [0.0, [0.0]*ArraySize, [0.0]*ArraySize]
IndexTime = 0
IndexHeight = 1
IndexVelocity = 2
```

The RK4 Time Step

To transform the RK4 code for the time step of the spring:

```
x_hat = State[IndexPosition] + (delta_t*F1_position)
```

into an equivalent code to compute an array of heights, as in the 1D wave equation, we would code:

```
# Position - compute F1
F1_position = [0.0]*ArraySize
for i in range(ArraySize):
    F1_position[i] = State[IndexVelocity][i]

# Position - Compute Euler estimate
x_hat = [0.0]*ArraySize
for i in range(ArraySize):
    x_hat[i] = State[IndexHeight][i] + (delta_t*F1_position[i])
```

Boundary Conditions (BC)

The last line of the previous code is a function to enforce boundary conditions. As indicated earlier, the most natural condition for the water waves is that they are reflected at the boundaries - homogeneous Neumann BC. Mathematically this means that $\frac{\partial u(x,t)}{\partial x} = 0$ at the spatial beginning and end for every t. The normal derivative is in this case the derivative with respect to x, with the appropriate sign. This is numerically enforced by making the beginning and end values equal to the contiguous ones, for each time step. This is represented by the code:

```
# Neumann boundary conditions
def EnforceBoundaryConditions(array):
    array[0] = array[1]
    array[-1] = array[-2]
```

If the BCs were Dirichlet, this function would set those values at the one imposed by the BC.

Vectorized Functions

In this function we have used a **copy utility**, to copy a part of an array to another part. This utility is a first simple example of **vectorized functions**, i.e., array functions.

```
def CopyArray(array_source, array_destination):
    for i in range(ArraySize):
        array_destination[i] = array_source[i]
```

To compute RK4, four approximations of the derivative are obtained, each based on the previous one. For this purpose, the different approximations have to be computed from the previous step.

Besides the already mentioned computation of temporal heights by means of vectorized functions, other examples are:

The estimation of temporal velocities:

```
v_hat = State[IndexVelocity] + (delta_t*F1_velocity) # ODE
```

uses vectorized functions this way:

```
v_hat = [0.0]*ArraySize # PDE
# Velocity - Compute Euler estimation
for i in range(ArraySize):
    v_hat[i] = State[IndexVelocity][i] + (delta_t*F1_velocity[i])
```

An Array of Initial Positions

The array of initial positions needs to be generated (one can assume that initial velocities are zero). Although a function could be used, this would be very little realistic. The code which follows generates random surface waves using some octaves of the so called Perlin noise. These positions are not kept while the program is running, to increase efficiency. A function `SetInitialState` is used, which can be called if a reset is needed.

```
def setup():
    # Set initial conditions
    noiseSeed(0)
    for i in range(ArraySize):
        worldX = 2341.17 + delta_x * ( float )i
        State[IndexHeight][i] = (0.5*anoise(worldX*0.0625) +
                                 0.4*anoise(worldX*0.125) +
                                 0.3*anoise(worldX*0.25) +
                                 0.2*anoise(worldX*0.5))
        State[IndexVelocity][i] = 0.0
    State[IndexTime] = 0.0

    # Enforce boundary conditions
    EnforceBoundaryConditions(State[IndexHeight])
    EnforceBoundaryConditions(State[IndexVelocity])
```

5.4.2 Final Stage: the Time Step

Recall the RK4 code for the elastic spring simulation:

```
# Time step function.
def TimeStep(delta_t):
    # Position - compute F1
    F1_position = State[IndexVelocity]

    # Velocity - compute F1
    F1_velocity = (-Stiffness/BobMass)*State[IndexPosition]

    # Position - Compute Euler estimate
    x_hat = State[IndexPosition] + (delta_t*F1_position)

    # Velocity - Compute Euler estimate
    v_hat = State[IndexVelocity] + (delta_t*F1_velocity)

    # Position - compute F2
    F2_position = v_hat

    # Velocity - compute F2
    F2_velocity = (-Stiffness/BobMass)*x_hat

    # Position - Compute Euler estimate
    x_hat_2 = State[IndexPosition] + (delta_t/2)*F2_position

    # Velocity - Compute Euler estimate
    v_hat_2 = State[IndexVelocity] + (delta_t/2)*F2_velocity

    # Position - compute F3
    F2_position = v_hat_2

    # Velocity - compute F3
    F2_velocity = (-Stiffness/BobMass)*x_hat_2

    # Position - Compute Euler estimate
    x_hat_3 = State[IndexPosition] + (delta_t)/2*F2_position

    # Velocity - Compute Euler estimate
    v_hat_3 = State[IndexVelocity] + (delta_t/2)*F2_velocity

    # Position - compute F4
    F2_position = v_hat_3

    # Velocity - compute F4
    F2_velocity = (-Stiffness/BobMass)*x_hat_3

    # Position - Update (RK4)
    State[IndexPosition] += (delta_t/6.0)*(F1_position + 2*F2_position + 2*F3_position +
     F4_position)

    # Velocity - Update (RK4)
    State[IndexVelocity] += (delta_t/6.0)*(F1_velocity + 2*F2_velocity + 2*F3_velocity +
     F4_velocity)

```

```
    # update time.
    State[IndexTime] += delta_t
```

For the waves simulation, the RK4 structure is similar, but the different elements mentioned above are used:

```
# Time Step function.
def TimeStep(delta_t):
    # Position - compute F1
    F1_position = [0.0]*ArraySize
    for i in range(ArraySize):
        F1_position[i] = State[IndexVelocity][i]

    # Velocity - compute F1
    uxx = approximate_uxx(State[IndexPosition])
    F1_velocity = [0.0]*ArraySize
    for i in range(ArraySize):
        F1_velocity[i] = sq(ConstantC)*uxx[i]

    # Position - compute Euler estimate
    x_hat = [0.0]*ArraySize
    for i in range(ArraySize):
        x_hat[i] = State[IndexPosition][i] + delta_t*F1_position[i]

    # Velocity - compute Euler estimate
    v_hat = [0.0]*ArraySize
    for i in range(ArraySize):
        v_hat[i] = State[IndexVelocity][i] + delta_t*F1_velocity[i]

    # Position - compute F2
    F2_position = [0.0]*ArraySize
    for i in range(ArraySize):
        F2_position[i] = v_hat[i]

    # Velocity - compute F2
    uxx_2 = approximate_uxx(x_hat)
    F2_velocity = [0.0]*ArraySize
    for i in range(ArraySize):
        F2_velocity[i] = sq(ConstantC)*uxx_2[i]

    # Position - compute Euler estimate
    x_hat_2 = [0.0]*ArraySize
    for i in range(ArraySize):
        x_hat_2[i] = State[IndexPosition][i] + (delta_t/2)*F2_position[i]

    # Velocity - compute Euler estimate
    v_hat_2 = [0.0]*ArraySize
    for i in range(ArraySize):
        v_hat_2[i] = State[IndexVelocity][i] + (delta_t/2)*F2_velocity[i]

    # Position - compute F3
    F3_position = [0.0]*ArraySize
    for i in range(ArraySize):
        F3_position[i] = v_hat_2[i]

    # Velocity - compute F3
    uxx_3 = approximate_uxx(x_hat_2)
    F3_velocity = [0.0]*ArraySize
    for i in range(ArraySize):
        F3_velocity[i] = sq(ConstantC)*uxx_3[i]

    # Position - compute Euler estimate
    x_hat_3 = [0.0]*ArraySize
    for i in range(ArraySize):
        x_hat_3[i] = State[IndexPosition][i] + (delta_t/2)*F3_position[i]

    # Velocity - compute Euler estimate
    v_hat_3 = [0.0]*ArraySize
    for i in range(ArraySize):
        v_hat_3[i] = State[IndexVelocity][i] + (delta_t/2)*F3_velocity[i]

    # Position - compute F4
    F4_position = [0.0]*ArraySize
    for i in range(ArraySize):
```

```
        F4_position[i] = v_hat_3[i]

    # Velocity - compute F4
    uxx_3 = approximate_uxx(x_hat_3)
    F4_velocity = [0.0]*ArraySize
    for i in range(ArraySize):
        F4_velocity[i] = sq(ConstantC)*uxx_3[i]

    # Position - Update (RK4)
    for i in range(ArraySize):
        State[IndexPosition][i] += (delta_t/6)*(F1_position[i]+2*F2_position[i]+2*
F3_position[i]+F4_position[i])
    # Velocity - Update (RK4)
    for i in range(ArraySize):
        State[IndexVelocity][i] += (delta_t/6)*(F1_velocity[i]+2*F2_velocity[i]+2*
F3_velocity[i]+F4_velocity[i])

    # Apply boundary conditions to height and velocity.
    EnforceBoundaryConditions(State[IndexPosition])
    EnforceBoundaryConditions(State[IndexVelocity])

    # Update time.
    State[IndexTime] += delta_t
```

Introducing Interactivity in the Visualization of the Simulation

The resulting code offers an interesting simulation, with seemingly correct approximate results. An interesting addition is to allow for user input. For instance, that the water height can be modified by mouse clicking. Mathematically this is called a **perturbation**, a small change in the conditions. A physcal perturbation would result from throwing a stone to the water surface. The following code allows for interactivity:

```
def GetInput():
    if (mousePressed && mouseButton == LEFT):
        mouseCellX = mouseX/float(PixelsPerCell)
        mouseCellY = (height/2-mouseY)/float(PixelsPerCell)
        simY = mouseCellY*delta_x

        # Find the position of the array where the mouse is
        i = int(floor(mouseCellX + 0.5))

        # Only act if within the window
        if (i > 0) and (i < ArraySize-1):
            State[IndexHeight][i] = simY
            State[IndexVelocity][i] = 0.0
```

The simulation becomes unstable after introducing interactivity: the perturbation is amplified. Implicit numerical methods, such as Crank-Nicolson, have better stability properties. To code this, one needs to consider the numerical solution of linear systems, to which we turn next.

5.5 Implementation of Jacobi's Method for the 1D Wave Equation

5.5.1 Data Structures

The matrix A of the linear system appearing in the numerical solution of the wave equation is **sparse**. A structure can be defined for this type of matrices, namely, A as an array of N **sparse rows** where a `Sparse Row` stores only non-zero elements.

For Jacobi's method implementation for the wave equation, this type of structures will not be needed, as the matrix is tridiagonal. Recall that the iteration can be simply expressed as:

$$h_i^{(n+1)} = \frac{b_i - \sum_{j \neq i} a_{ij} h_j^{(n)}}{a_{ii}}$$

Only the diagonal and two off-diagonals are non-zero. Only these three N-dimensional diagonal vectors need to be stored to compute the $h_j^{(n+1)}$, and the previous $h_j^{(n)}$.

5.5.2 Implementation of the Implicit Method

Recall that the implicit iterative scheme for the 1D wave equation is:

$$-\mu h_{j+1}^{m+1} + (1+2\mu)h_j^{m+1} - \mu h_{j-1}^{m+1} = 2h_j^m - h_j^{m-1}$$

which represents the system of linear equations $Ah = b$ where:

$$A = \begin{bmatrix} 1+2\mu & -\mu & & & & & \cdots \\ -\mu & 1+2\mu & -\mu & & & & \cdots \\ & -\mu & 1+2\mu & -\mu & & & \cdots \\ \vdots & & \ddots & \ddots & \ddots & & \cdots \\ \cdots & & & -\mu & 1+2\mu & -\mu & \\ \cdots & & & & -\mu & 1+2\mu & -\mu \\ \cdots & & & & & -\mu & 1+2\mu \end{bmatrix}$$

$$h = \begin{bmatrix} h_1^{m+1} \\ h_2^{m+1} \\ h_3^{m+1} \\ \vdots \\ h_{N-4}^{m+1} \\ h_{N-3}^{m+1} \\ h_{N-2}^{m+1} \end{bmatrix} \qquad b = \begin{bmatrix} 2h_1^m - h_1^{m-1} \\ 2h_2^m - h_2^{m-1} \\ 2h_3^m - h_3^{m-1} \\ \vdots \\ 2h_{N-4}^m - h_{N-4}^{m-1} \\ 2h_{N-3}^m - h_{N-3}^{m-1} \\ 2h_{N-2}^m - h_{N-2}^{m-1} \end{bmatrix}$$

Jacobi's iteration is thus:

$$h_j^{m+1,(n+1)} = \frac{2h_j^m - h_j^{m-1}}{(1+2\mu)} + \frac{\mu}{(1+2\mu)}(h_{j+1}^{m+1,(n)} + h_{j-1}^{m+1,(n)})$$

Computation of the Jacobi solver iterations

As the independent terms are the same for each iteration to compute the numerical solution of the linear system, one should compute them only once:

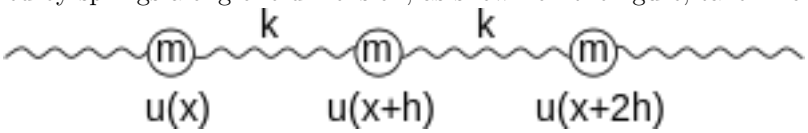
```
# Assuming 20 iterations
maxIterations = 20

# Data structure to store info about u^(m+1),(n)
heightCurrent = [0.0]*ArraySize
for i in range(ArraySize): # Initialize with u^m
    heightCurrent[i] = State[IndexHeight][i]

# Data structure to store info about u^(m+1),(n+1)
heightNext = [0.0]*ArraySize

# Compute iterations
for iteration in range(maxIterations):
    # Iterative loop: apply Jacobi formula to calculate each cell of next x
    for i in range(1,ArraySize-1):
        aij = -sq(ConstantC)*sq(delta_t)/sq(delta_x)
        aii = 1.0 + (2.0*sq(ConstantC)*sq(delta_t)/sq(delta_x))

        # Compute the summatory term
        summatory = -aij*(heightCurrent[i-1]+heightCurrent[i+1])

        # Compute the bi term
        bi = 2*State[IndexHeight][i] - State_Previous[IndexHeight][i]

        # Compute u^(m+1),(n+1) from the previous terms
        heightNext[i] = (bi-summatory)/aii

    # Post-iteration: apply boundary conditions
    EnforceBoundaryConditions(heightNext)

    # Update u^{(m+1),(n)} with the obtained u^{(m+1),(n+1)}
    for i in range (ArraySize):
        heightCurrent[i] = heightNext[i]
```

NB: Christopher Horvath's blog uses independent terms slightly different from these ones, as the approximation used is slightly different.

5.6 Supplementary Material: Model and Analytical Solution of the 1D Wave Equation

5.6.1 The Physical Model of the Wave Equation

One of the derivations of the wave equation comes from considering an array of small mass points m linked by springs along one dimension, as shown on the figure, taken from Wikipedia:



Let us consider the equation verified by the mass at $x+h$, where $u(x+h,t)$ denotes the displacement wrt the equilibrium position. One has the usual force:

$$F_{newton} = m\frac{\partial^2}{\partial t^2}u(x+h,t)$$

while the displacements verify Hooke's law due to the elasticity of the springs:

$$F_{hooke} = k\left[u(x+2h,t) - u(x+h,t)\right] - k\left[u(x+h,t) - u(x,t)\right]$$

Equating both results:

$$m\frac{\partial^2}{\partial t^2}u(x+h,t) = k\left[u(x+2h,t) - 2u(x+h,t) + u(x,t)\right]$$

A continuous model derives N masses linked by N springs along a fixed length $L = Nh$. The total mass is $M = Nm$; let $K = \frac{k}{N}$. Then the previous equation becomes:

$$\frac{\partial^2}{\partial t^2}u(x+h,t) = \frac{KL^2}{M}\frac{u(x+2h,t) - 2u(x+h,t) + u(x,t)}{h^2}$$

Taking the limit $h \to 0$:

$$\frac{\partial^2 u(x,t)}{\partial t^2} = \frac{KL^2}{M}\frac{\partial^2 u(x,t)}{\partial x^2}$$

and defining $c^2 = \frac{KL^2}{M}$, the 1 D wave equation is obtained:

$$\frac{\partial^2 u(x,t)}{\partial t^2} = c^2\frac{\partial^2 u(x,t)}{\partial x^2}$$

which we usually write in the more compact form $u_{tt}(x,t) = c^2 u_{xx}(x,t)$.

5.6.2 Analytic Solutions of the 1D Wave Equation

Solution 1: Waves with Speed c

The change of variables $\xi = x - ct$, $\eta = x + ct$ transforms the wave equation $u_{tt} = c^2 u_{xx}$ into $\frac{\partial^2 u}{\partial \xi \partial \eta} = 0$, as using the chain rule we have:

$$\frac{\partial^2 u}{\partial \xi \partial \eta} = \frac{u_{xx}}{4} - \frac{u_{tt}}{4c^2} = 0$$

To solve $\frac{\partial^2 u}{\partial \xi \partial \eta} = 0$, let $v = \frac{\partial u}{\partial \eta}$. Then we have $\frac{\partial v}{\partial \xi} = 0$, whose general solution is $v = G(\eta)$. This means that $v = \frac{\partial u}{\partial \eta} = G(\eta)$, which integrating leads to $u = \phi(\xi) + \psi(\eta)$, for any ϕ and ψ. The **general solution** of the wave equation is thus

$$u(x,t) = \phi(x - ct) + \psi(x + ct)$$

If we denote $u_1(x,t) = \phi(x-ct)$, then $u_1(x,0) = \phi(x)$ and $u_1(x+ct,t) = \phi(x+ct-ct) = \phi(x)$. Thus, at time t we find the same values at a position displaced by ct. It is said that we have waves travelling with speed c.

If the initial position and velocity are $u(x,0) = f(x)$ and i $u_t(x,0) = g(x)$, then the **particular solution** verifying these conditions is given by **d'Alembert's formula**:

$$u(x,t) = \frac{f(x-ct)+f(x+ct)}{2} + \frac{1}{2c}\int_{x-ct}^{x+ct} g(s)ds$$

as

$$f(x) = u(x,0) = \phi(x) + \psi(x) \qquad g(x) = u_t(x,0) = -c\phi'(x) + c\psi'(x)$$

and integrating the latter equation

$$\int_a^x g(s)ds = -c\phi(x) + c\psi(x) \qquad \frac{1}{c}\int_a^x g(s)ds = -\phi(x) + \psi(x)$$

and thus:

$$\psi(x) = \frac{1}{2}f(x) + \frac{1}{2c}\int_a^x g(s)ds \qquad \phi(x) = \frac{1}{2}f(x) - \frac{1}{2c}\int_a^x g(s)ds$$

which leads to

$$u(x,t) = \phi(x-ct) + \psi(x+ct) = \frac{1}{2}f(x-ct) - \frac{1}{2c}\int_a^{x-ct} g(s)ds = \frac{1}{2}f(x+ct) + \frac{1}{2c}\int_a^{x+ct} g(s)ds$$

from which the expression of the particular solution results.

The idea for the change of variables comes from an algebraic reformulation of the 1D wave equation

$$\frac{\partial^2 u}{\partial t^2} - c^2\frac{\partial^2 u}{\partial x^2} = 0$$

as sum times difference giving the difference of squares:

$$\left[\frac{\partial}{\partial t} + c\frac{\partial}{\partial x}\right]\left[\frac{\partial}{\partial t} - c\frac{\partial}{\partial x}\right]u = 0$$

and thus

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0 \quad o \quad \frac{\partial u}{\partial t} - c\frac{\partial u}{\partial x} = 0$$

suggesting the change of variables used$\xi = x - ct$, $\eta = x + ct$.

Solution 2 by Separation of Variables

The wave equation is **linear** as the linear combination of two solutions $v(x,t)$ and $w(x,t)$, $\lambda v(x,t) + \mu w(x,t)$, is a solution as well. Thus a strategy to find the general solution is to look for the subspace of functions which are solutions. We try solutions $u(x,t)$ which can be expresse as the product of two functions of x and t of the form $u(x,t) = F(x)G(t)$. The 1D wave equation leads to:

$$F(x)\frac{d^2G(t)}{dt^2} = c^2G(t)\frac{d^2F(x)}{dx^2}$$

$$\frac{1}{G(t)}\frac{d^2G(t)}{dt^2} = c^2\frac{1}{F(x)}\frac{d^2F(x)}{dx^2}$$

As the LHS and RHS are functions of different variables, the only possibility is that they are constant:

$$\frac{1}{G(t)}\frac{d^2G(t)}{dt^2} = c^2\frac{1}{F(x)}\frac{d^2F(x)}{dx^2} = \omega$$

and thus,

$$\frac{d^2G(t)}{dt^2} = \omega G(t) \qquad \frac{d^2F(x)}{dx^2} = \frac{\omega}{c^2}F(x)$$

Both equations represent **eigenfunction** problems for a linear operator: the second derivative. This is similar to the spring equation. We have solutions such as:

$$G(t) = e^{-i\sqrt{\omega}t} \qquad F(x) = e^{-i\frac{\sqrt{\omega}}{c}x}$$

which leads to Fourier expressions for the particular solution.

Exercise 24
*Try separation of variables to find the general solution of the 1D heat equation $u_t(x,t) = cu_{xx}(x,t)$. This equation is also known as the **diffusion equation**, as it represents that heat diffuses from positions at higher temperatures to those at lower ones; c is the thermal conductivity coefficient.*

Chapter 6

2D+t PDEs

6.1 Discretization and Numerical Schemes for the 2D+t Heat Equation

6.1.1 The 2D+t Heat Equationn

A very general way of expressing the heat/diffusion PDE is $u_t = \text{div}(c\nabla u)$, where c is the diffusion coefficient; ∇ is the (gradient) differential operator, which applied on the scalar function of two space variables $u(x, y, t)$ results into the 2D vector $\nabla u = (u_x, u_y)^T$; and *div* is the (divergence) differential operator, which on a 2-component vector function $f : \Omega \to I\!R^2$, $\bar{f}(x, y) = (v(x, y), w(x, y))$ is defined as $\text{div}(\bar{f}) = v_x + w_y$.

If c is constant, the **2D+t heat equation** can be written as

$$u_t = c\nabla^2 u = c(u_{xx} + u_{yy})$$

One should add the **initial condition**, $u(x, y, 0) = u_0(x, y)$, for every (x, y) in the 2D region considered, at the initial time $t = 0$; and the **boundary conditions**, Dirichlet or Neumann. In 1D, the region is usually a segment, and its boundary is made of the initial and end points of the segment. In 2D, the region can have very diverse shapes. If the region is a circle, the boundary is its exterior circumference. If the region is a rectangle, the boundary is made of its four sides. The boundary conditions, as usual, are applied $\forall(x, y) \in Boundary$ and $\forall t > 0$.

6.1.2 Discretization

In 2D we denote $u(i\Delta x, j\Delta y, m\Delta t) \simeq u^m_{i,j}$. Approximations of the derivatives similar as those in the case of one space dimension are:

$$\frac{\partial u(i\Delta x, j\Delta y, m\Delta t)}{\partial t} \simeq \frac{u(i\Delta x, j\Delta y, (m+1)\Delta t) - u(i\Delta x, j\Delta y, m\Delta t)}{\Delta t} \simeq \frac{u^{m+1}_{i,j} - u^m_{i,j}}{\Delta t}$$

and

$$c(u_{xx}(i\Delta x, j\Delta y, m\Delta t) + u_{yy}(i\Delta x, j\Delta y, m\Delta t)) \simeq$$

$$c(\frac{u^m_{i+1,j} - 2u^m_{i,j} + u^m_{i-1,j}}{\Delta x^2} + \frac{u^m_{i,j+1} - 2u^m_{i,j} + u^m_{i,j-1}}{\Delta y^2})$$

6.1.3 Explicit Formulation

For simplicity, we assume $\Delta x = \Delta y$; then equating both approximations, the discretized 2D heat equation is:

$$\frac{u^{m+1}_{i,j} - u^m_{i,j}}{\Delta t} = \frac{c}{\Delta x^2}(u^m_{i+1,j} + u^m_{i,j+1} - 4u^m_{i,j} + u^m_{i-1,j} + u^m_{i,j-1})$$

Isolating the $(m + 1)$ term, the **explicit iterative scheme** results:

$$u^{m+1}_{i,j} = \mu u^m_{i+1,j} + \mu u^m_{i,j+1} + (1 - 4\mu)u^m_{i,j} + \mu u^m_{i-1,j} + \mu u^m_{i,j-1}$$

where we use $\mu = \frac{c\Delta t}{\Delta x^2}$. Let us remark that the array which needs to be computed at each iteration is approximately NxN.

6.1.4 Implicit Formulations

For 2D+t, the **implicit** and **Crank-Nicolson** formulations can be obtained in a way similar to the 1D+t heat equation.

An **implicit method** derives from:

$$\frac{u_{i,j}^{m+1} - u_{i,j}^{m}}{\Delta t} = \frac{c}{\Delta x^2}(u_{i+1,j}^{m+1} + u_{i,j+1}^{m+1} - 4u_{i,j}^{m+1} + u_{i-1,j}^{m+1} + u_{i,j-1}^{m+1})$$

which leads to:

$$-\mu u_{i+1,j}^{m+1} - \mu u_{i,j+1}^{m+1} + (1+4\mu)u_{i,j}^{m+1} - \mu u_{i-1,j}^{m+1} - \mu u_{i,j-1}^{m+1} = u_{i,j}^{m}$$

where the linear system contains approximately NxN equations and NxN unknowns; Jacobi or Gauss - Seidel can be used to obtain its numerical solution.

A **Crank-Nicolson** method derives from:

$$\frac{u_{i,j}^{m+1} - u_{i,j}^{m}}{\Delta t} = \frac{c}{2\Delta x^2}(u_{i+1,j}^{m} + u_{i,j+1}^{m} - 4u_{i,j}^{m} + u_{i-1,j}^{m} + u_{i,j-1}^{m}) +$$

$$+\frac{c}{2\Delta x^2}(u_{i+1,j}^{m+1} + u_{i,j+1}^{m+1} - 4u_{i,j}^{m+1} + u_{i-1,j}^{m+1} + u_{i,j-1}^{m+1})$$

Using $\mu = \frac{c\Delta t}{2\Delta x^2}$, it results:

$$-\mu u_{i+1,j}^{m+1} - \mu u_{i,j+1}^{m+1} + (1+4\mu)u_{i,j}^{m+1} - \mu u_{i-1,j}^{m+1} - \mu u_{i,j-1}^{m+1} =$$

$$= \mu u_{i+1,j}^{m} + \mu u_{i,j+1}^{m} + (1-4\mu)u_{i,j}^{m} + \mu u_{i-1,j}^{m} + \mu u_{i,j-1}^{m}$$

i.e., a system of approximately NxN linear equations with NxN unknowns must be solved at each iteration.

6.1.5 Initial and Boundary Conditions

From the *initial condition* $u(x,y,0) = u_0(x,y)$ one obtains $u_{i,j}^0 = u(i\Delta x, j\Delta y, 0) = u_0(i\Delta x, j\Delta y)$ with $0 \le i,j \le N$, values which are used in the first iteration.

If the region is a rectangle, the boundary is composed of its four sides, represented by $i = 0$ (for all j), $i = N$ (for all j), $j = 0$ (for all i) and $j = N$ (for all i). A Dirichlet boundary condition fixes the values, for all $t > 0$, i.e., $\forall m$, of $u_{0,j}^m \forall j$, $u_{N,j}^m \forall j$, $u_{i,0}^m \forall i$, $u_{i,N}^m \forall i$. Homogeneous Neumann boundary conditions would be $u_x(0\Delta x, j\Delta y, m\Delta t) = 0 = u_x(N\Delta x, j\Delta y, m\Delta t), \forall j, \forall m$ for $i = 0$ and $u_y(i\Delta x, 0\Delta y, m\Delta t) = 0 = u_y(i\Delta x, N\Delta y, m\Delta t), \forall i, \forall m$.

As in the 1D+t case, when using implicit methods, applying the boundary conditions requires to modify some of the equations in the system of linear equations to be solved at each iteration; for instance, in the case of a rectangular boundary that we are considering, 4N equations need to be modified.

6.2 2D+t Heat Equation for Image Processing

6.2.1 Images

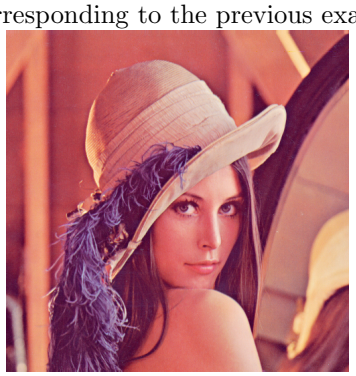
An image can be formally defined as a function $u(x,y) : \Omega \to I\!R$ where $\Omega \subset I\!R^2$ is the domain where the image is defined, usually a rectangle. If the image is represented on a screen, $(x,y) \in \Omega$ can be considered as representing the pixels positions, while the scalar values $u(x,y)$ are the grey levels (light intensity) at each pixel. In the mathematical representation *black* is $u(x,y) = 0$ and *white* $u(x,y) = 1$. Each intensity or grey level between black and white will be a value within the interval $[0,1]$.

In a colour image, $u(x,y)$ is usually an $I\!R^3$ vector instead of a scalar. It is made of the levels of *R red*, *G green* and *B blue*. This is in the *RGB* colour representation system; there are other colour representation systems, such as *HSL* (*Hue*, *Saturation*, *Lightness*).

6.2.2 Noise in Images

The images received through a communication channel are usually **noisy**, i.e., they contain random perturbations. Indeed, the name noise stems from the audible perception of the perturbations in old AM radios. Noise can be generated in the communication, but can have different sources; for instance, the sensors of current digital cameras are susceptible of random perturbations. The noisy image can be represented as $u_0(x, y)$ resulting from the contamination of an image $u(x, y)$ by additive noise $\nu(x, y)$, i.e., the noisy image is $u_0(x, y) = u(x, y) + \nu(x, y)$. The following is an example of noisy image:

Most receiver systems process the images to remove the noise, in order to provide a nice image. The process of obtaining $u(x, y)$ from $u_0(x, y)$ is called **image denoising**. The image without noise corresponding to the previous example is:



6.2.3 Using the Heat Equation to Denoise Images

PDEs are often used to process images. The processing realizes a known property of the PDE used. Indeed, a property of the heat equation is that it eliminates noise, and it is used with this purpose. In signal processing terminology, the process is called **filtering**, in this case we can say that the heat equation is used to filter the noise.

More precisely we assume that $u(x, y, t)$, the solution at time t of the heat PDE with initial condition $u(x, y, 0) = u_0(x, y)$, is an image where the noise in the initial image has been eliminated at some time t.