INTEGRATION OF ORDINARY DIFFERENTIAL EQUATIONS

In this chapter, we discuss the integration of Ordinary Differential Equations (ODEs).

Color code for the chapter:

- Definitions and important results are given in **pink** boxes.
- Skeletons of algorithms are in **purple** boxes.
- Examples are given in **green** boxes.
- Remarks and notions that can be skipped for the first read are in **black** boxes.
- Links are emphasized in **blue**.

Before starting, we want to stress that there is a full zoo of methods to integrate ODEs (some of them are implemented in Python, see the documentation page of `scipy.integrate`). It is impossible and useless to list them all. We will present some of them which are particularly relevant to understand key concepts (convergence of an integrator, consistency of an integrator, symplectic integrator, explicit or implicit integrator, stability of an integrator, etc.) and which are heavily used in Physics.

We start by discussing the case of first-order ODEs and Initial Value Problems (IVPs), namely,

$$\begin{cases} \dfrac{\mathrm{d}\vec{x}}{\mathrm{d}t} = f(\vec{x}, t), \\ \vec{x}(t_\mathrm{i}) = \vec{x}_\mathrm{i}, \end{cases} \tag{1}$$

where $\vec{x} \in \mathbb{R}^d$ is a d-dimensional vector, and where $f : \mathbb{R}^d \times \mathbb{R} \to \mathbb{R}^d$. To be clear, $\vec{x} = (x_1, x_2, \ldots, x_d)$ and $f(\vec{x}, t) = (f_1(\vec{x}, t), f_2(\vec{x}, t) \ldots, f_d(\vec{x}, t))$. The case of ODEs of larger order, as well as Boundary Value Problems (BVPs) will be discussed at the end of this chapter.

I. General method to construct a numerical integrator of ODEs

1. Introduction

The basic idea behind the integration of ODEs is to discretize in time the Cauchy problem (1). Imagine that you want to integrate in the time interval $[t_\mathrm{i},\, t_\mathrm{f}]$ with N steps of size $h = (t_\mathrm{f} - t_\mathrm{i})/(N-1)$. Your objective is to compute $\overrightarrow{x_1}^{(h)}, \dots, \overrightarrow{x_n}^{(h)}, \dots, \overrightarrow{x_N}^{(h)}$ such that $\overrightarrow{x_n}^{(h)}$ is a good approximation of the exact solution $\overrightarrow{x_\mathrm{e}}(t)$ evaluated at time $t_n = t_\mathrm{i} + nh$, $\overrightarrow{x_\mathrm{e}}(t_n)$. Different algorithms come from different choices of the time discretization of the derivative. This is discussed in the next paragraph.

2. Discretization of the time derivative

a. One-step methods

The idea is to perform Taylor expansion of the exact solution, and then to assume that the numerical integrator is convergent, namely, that it approximates well the solution (see below for a proper definition). For instance, a first-order Taylor expansion gives

$$\overrightarrow{x_\mathrm{e}}(t_n + h) = \overrightarrow{x_\mathrm{e}}(t_n) + h\frac{\mathrm{d}\overrightarrow{x_\mathrm{e}}}{\mathrm{d}t}(t_n) + \frac{1}{2}h^2\frac{\mathrm{d}^2\overrightarrow{x_\mathrm{e}}}{\mathrm{d}t^2}(t_n) + O(h^3) = \overrightarrow{x_\mathrm{e}}(t_n) + hf(\overrightarrow{x_\mathrm{e}}(t_n), t_n) + O(h^2). \tag{2}$$

Then, if we assume that the method is convergent, $\overrightarrow{x_\mathrm{e}}(t_n + h) = \overrightarrow{x_{n+1}}^{(h)}$ and $\overrightarrow{x_\mathrm{e}}(t_n) = \overrightarrow{x_n}^{(h)}$ and we eventually get the following time discretization of Eq. (1):

$$\overrightarrow{x_{n+1}}^{(h)} = \overrightarrow{x_n}^{(h)} + hf(\overrightarrow{x_n}, t_n).$$

This is called the **Forward Euler method** because the function f (*i.e.*, the slope of the solution) is evaluated at the previous time step (see Fig. 1). This method will be discussed below.

But we could have done the Taylor expansion the other way around, namely,

$$\overrightarrow{x_\mathrm{e}}(t_n) = \overrightarrow{x_\mathrm{e}}(t_n + h) - h\frac{\mathrm{d}\overrightarrow{x_\mathrm{e}}}{\mathrm{d}t}(t_n + h) + O(h^2) = \overrightarrow{x_\mathrm{e}}(t_n + h) - hf(\overrightarrow{x_\mathrm{e}}(t_n + h), t_n + h) + O(h^2), \tag{3}$$

leading to the time discretization

$$\overrightarrow{x_{n+1}}^{(h)} = \overrightarrow{x_n}^{(h)} + hf(\overrightarrow{x_{n+1}}, t_n + h).$$

This is called the **Backward Euler method** because the function f is evaluated at the next time step (see Fig. 1). This method wil also be discussed below.

We could also consider a point between t_n and $t_n + h$ as a reference to perform the Taylor expansion, for instance the midpoint $t_n + h/2$. Then, the Taylor expansions are

$$\overrightarrow{x_\mathrm{e}}(t_n) = \overrightarrow{x_\mathrm{e}}(t_n + h/2) - \frac{h}{2}\frac{\mathrm{d}\overrightarrow{x_\mathrm{e}}}{\mathrm{d}t}(t_n + h/2) + \frac{1}{2}\left(\frac{h}{2}\right)^2\frac{\mathrm{d}^2\overrightarrow{x_\mathrm{e}}}{\mathrm{d}t^2}(t_n + h/2) + O(h^3)$$

and

$$\overrightarrow{x_\mathrm{e}}(t_n + h) = \overrightarrow{x_\mathrm{e}}(t_n + h/2) + \frac{h}{2}\frac{\mathrm{d}\overrightarrow{x_\mathrm{e}}}{\mathrm{d}t}(t_n + h/2) + \frac{1}{2}\left(\frac{h}{2}\right)^2\frac{\mathrm{d}^2\overrightarrow{x_\mathrm{e}}}{\mathrm{d}t^2}(t_n + h/2) + O(h^3),$$

such that

$$\overrightarrow{x_\mathrm{e}}(t_n + h) = \overrightarrow{x_\mathrm{e}}(t_n) + h\frac{\mathrm{d}\overrightarrow{x_\mathrm{e}}}{\mathrm{d}t}(t_n + h/2) + O(h^3). \tag{4}$$

However, the value of the derivative at time $t_n + h/2$ is not known so we need to express it as a function of the values at times t_n and $t_n + h$, thanks to two other Taylor expansions:

$$\begin{cases} \dfrac{\mathrm{d}\overrightarrow{x_\mathrm{e}}}{\mathrm{d}t}(t_n) = \dfrac{\mathrm{d}\overrightarrow{x_\mathrm{e}}}{\mathrm{d}t}(t_n + h/2) - \dfrac{h}{2}\dfrac{\mathrm{d}^2\overrightarrow{x_\mathrm{e}}}{\mathrm{d}t^2}(t_n + h/2) + O(h^2), \\ \dfrac{\mathrm{d}\overrightarrow{x_\mathrm{e}}}{\mathrm{d}t}(t_n + h) = \dfrac{\mathrm{d}\overrightarrow{x_\mathrm{e}}}{\mathrm{d}t}(t_n + h/2) + \dfrac{h}{2}\dfrac{\mathrm{d}^2\overrightarrow{x_\mathrm{e}}}{\mathrm{d}t^2}(t_n + h/2) + O(h^2), \end{cases}$$

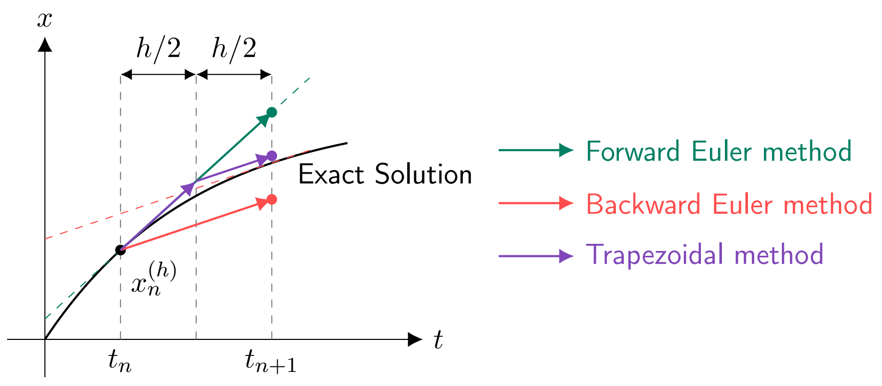

Figure 1: **Illustration of three different methods to integrate an ODE.** We show the behavior for one time step of the Forward Euler method (in green), the Backward Euler method (in pink) and the Trapezoidal method (in purple). The exact solution is represented as a black continuous line, and the dashed lines mark the tangent lines to the exact solution at times t_n and t_{n+1}. The black point marks the numerical solution at step n, and the different colored points mark the numerical solution at step $n+1$ for the different integration methods.

such that

$$\frac{\mathrm{d}\vec{x_\mathrm{e}}}{\mathrm{d}t}(t_n+h/2)=\frac{1}{2}\left[\frac{\mathrm{d}\vec{x_\mathrm{e}}}{\mathrm{d}t}(t_n)+\frac{\mathrm{d}\vec{x_\mathrm{e}}}{\mathrm{d}t}(t_n+h)\right]+O(h^2). \tag{5}$$

We eventually obtain the following time discretization:

$$\overrightarrow{x_{n+1}}^{(h)}=\vec{x_n}^{(h)}+\frac{h}{2}\left[f(\vec{x_n}^{(h)},t_n)+f(\overrightarrow{x_{n+1}}^{(h)},t_n+h)\right].$$

This is called the **Trapezoidal method** because the function f is evaluated at the previous time step and at the next time step (see Fig. 1). This method will be discussed below.

b. Multi-step methods

So far, we have considered time discretizations which only involve the knowledge of the solution at the previous step. These methods are called one-step methods. However, there are schemes which require the knowledge of the solution at the two previous steps or more (multi-step methods). We provide below an example. We again start from the Taylor expansion of $\vec{x_\mathrm{e}}$ at time t_n+h, see Eq. (2). We then use a Taylor expansion of the time derivative of $\vec{x_\mathrm{e}}$ at time t_n-h to express the second derivative:

$$\begin{aligned}\frac{\mathrm{d}\vec{x_\mathrm{e}}}{\mathrm{d}t}(t_n-h)&=\frac{\mathrm{d}\vec{x_\mathrm{e}}}{\mathrm{d}t}(t_n)-h\frac{\mathrm{d}^2\vec{x_\mathrm{e}}}{\mathrm{d}t^2}(t_n)+O(h^2)\\ \Longrightarrow h\frac{\mathrm{d}^2\vec{x_\mathrm{e}}}{\mathrm{d}t^2}(t_n)&=f(\vec{x_\mathrm{e}}(t_n),t_n)-f(\vec{x_\mathrm{e}}(t_n-h),t_n-h)+O(h^2).\end{aligned}$$

This eventually leads to

$$\vec{x_\mathrm{e}}(t_n+h)=\vec{x_\mathrm{e}}(t_n)+\frac{3h}{2}f(\vec{x_\mathrm{e}}(t_n),t_n)-\frac{h}{2}f(\vec{x_\mathrm{e}}(t_n-h),t_n-h)+O(h^3),$$

from which we deduce the following time discretization:

$$\overrightarrow{x_{n+1}}^{(h)}=\vec{x_n}^{(h)}+\frac{h}{2}\left[3f(\vec{x_n}^{(h)},t_n)-f(\overrightarrow{x_{n-1}}^{(h)},t_n-h)\right].$$

This is called the Adams-Bashforth method.

c. Conclusion

Depending on the time discretization for the derivatives, you can construct a large family of numerical integrators of ODEs. The starting point is always a Taylor expansion of the exact solution at different time steps (your creativity is the limit!).

General definition of a numerical integrator
A numerical integrator of an ODE of the form given by Eq. (1) is a relation giving the numerical estimate of the solution at step $n+1$ as a function of the numerical estimates of the solution in the present and in the k past steps:

$$G(\overrightarrow{x_{n+1}}^{(h)}, \overrightarrow{x_n}^{(h)}, \overrightarrow{x_{n-1}}^{(h)}, \dots, \overrightarrow{x_{n-k}}^{(h)}, t_n) = \vec{0}. \tag{6}$$

The method is **explicit** if the numerical estimate of the solution at step $n+1$ only depends on the numerical estimates in the past. It can then be written as

$$\overrightarrow{x_{n+1}} = H(\overrightarrow{x_n}^{(h)}, \overrightarrow{x_{n-1}}^{(h)}, \dots, \overrightarrow{x_{n-k}}^{(h)}, t_n) \tag{7}$$

for a k-step method.
The method is **implicit** if the numerical estimate of the solution at step n depends on the numerical estimates in the past and in the present.

II. The Forward Euler method

In this section, we discuss the Forward Euler method introduced above.

1. Definition

The Forward Euler method is defined as follows.

Definition of the Forward Euler method
For the IVP given by Eq. (1), the Forward Euler method is a numerical integrator of ODEs given by the following recurrence relation:

$$\overrightarrow{x_{n+1}}^{(h)} = \overrightarrow{x_n} + hf(\overrightarrow{x_n}, t_n), \tag{8}$$

for $t_n = t_\mathrm{i} + nh$ and h the step size.

The Forward Euler method is **explicit**. Therefore, we can propose a simple algorithmic representation.

Algorithm for the Forward Euler method

```
$x \leftarrow x_\mathrm{i}$                          ▷ Initial condition.
$t \leftarrow t_\mathrm{i}$                          ▷ Initial time.
STORE($x$)                                ▷ Store the initial value of $x$.
STORE($t$)                                ▷ Store the initial value of $t$.
while $t < t_\mathrm{f}$ do
    $x \leftarrow x + hf(x,t)$               ▷ Update the solution.
    $t \leftarrow t + h$                     ▷ Update the time.
    STORE($x$)                            ▷ Store the current value of $x$.
    STORE($t$)                            ▷ Store the current value of $t$.
```

We discuss the accuracy of the method in the next section.

2. Local truncation error and consistency

We come back to the Taylor series at the root of the Forward Euler method, see Eq. (2). You can see that we have truncated all terms of order h^2 and above to define the Forward Euler method, see Eq. (8). We thus say that the **local truncation error** is $O(h^2)$.

Local truncation error
The **local truncation error** τ_n at step n is defined as the difference between $\overrightarrow{x_n}^{(h)}$ (estimated numerical value of the solution at time t_n) and $\overrightarrow{x_\mathrm{e}}(t_n)$ (exact value of the solution at time t_n), assuming that no error

was made in the previous steps. In other words,

$$\tau_n = \|\overrightarrow{x_n}^{(h)} - \overrightarrow{x_\mathrm{e}}(t_n)\| \text{ assuming } \overrightarrow{x_k}^{(h)} = \overrightarrow{x_\mathrm{e}}(t_k)\ \forall k < n. \tag{9}$$

From the local truncation error, we define the consistency of a numerical integrator.

Consistency of a numerical integrator
A numerical integrator of ODEs is **consistent** if the time discretization approximates the ODE well. Said differently, the exact solution to an ODE should verify the recurrence relation up to terms of order h^q with h the step size and $q > 1$ in the limit $h \to 0$.

Mathematically, a numerical integration is consistent if its local truncation error τ_n goes to 0 when the step size h goes to 0 at least as h^1, or equivalently, if

$$\forall n \in [\![1,\, N]\!]\ \lim_{h\to 0} \frac{\tau_n}{h} = 0. \tag{10}$$

From the above definitions, we can deduce some properties of the Forward Euler method.

Consistency of the Forward Euler method
The Forward Euler method has a local truncation error $\tau_n = O(h^2)$: it is therefore **consistent**.

We end this section by stressing that **the consistency does not tell you whether the numerical integrator gives you a good estimate of the solution of an ODE.** Instead, it just tells you that the Cauchy problem is well captured by the numerical integrator. Whether the numerical integrator gives a reasonable estimate of the solution is called **convergence**, and is discussed in the next section.

3. Global truncation error and convergence

Of course, when you integrate an ODE on a interval $[t_\mathrm{i},\, t_\mathrm{f}]$, local truncation errors accumulate. Therefore, a better assessment of the accuracy of a numerical integrator is given by looking at the **global truncation error**.

Global truncation error
The **global truncation error** e_n at step n is defined as the difference between $\overrightarrow{x_n}^{(h)}$ (estimated numerical value of the solution at time t_n) and $\overrightarrow{x_\mathrm{e}}(t_n)$ (exact value of the solution at time t_n), taking into account all possible errors in the previous steps. In other words,

$$e_n = \|\overrightarrow{x_n}^{(h)} - \overrightarrow{x_\mathrm{e}}(t_n)\|. \tag{11}$$

From the global truncation error, we define the convergence of a numerical integrator.

Convergence of a numerical integrator
A numerical integrator of ODEs is **convergent** if the numerical solution approaches the exact solution at any time when the step size h goes to 0.

Mathematically, a numerical integrator is convergent if

$$\forall t \in [t_\mathrm{i},\, t_\mathrm{f}]\ \lim_{\substack{h\to 0\\ n\to +\infty\\ nh = t - t_\mathrm{i}}} e_n = 0. \tag{12}$$

The way e_n goes to 0 when we take the two limits $h \to 0$ and $n \to +\infty$ but with $nh < +\infty$ makes it possible to define **the order** of the numerical integrator: the method is of **order** p (with $p > 0$) if $e_n = O(h^p)$ when $h \to 0$, $n \to +\infty$, $nh < +\infty$.

The convergence property is, of course, a property that you absolutely need in order to integrate an ODE in Physics.

We now illustrate the convergence property on the Forward Euler method. It is possible to prove that the Forward Euler method is of order 1, see D. F. Griffiths and D. J. Higham, *Numerical Methods for Ordinary Differential Equations*. However, the proof is complicated, and we just give an example below.

Imagine that you want to solve numerically the Cauchy problem

$$\frac{\mathrm{d}x}{\mathrm{d}t} = x, \quad x(0) = 1,$$

on $[0,\, 1]$. We know that the exact solution is $x_{\mathrm{e}}(t) = e^t$.

Now, let's look at the Forward Euler method: $x_{n+1}^{(h)} = x_n^{(h)} + hx_n^{(h)} = x_n^{(h)}(1+h)$. This is a geometric progression, and this implies that $x_n^{(h)} = (1+h)^n$. From this, we can compute the global truncation error:

$$e_n = \left|(1+h)^n - e^{nh}\right| = \left|e^{n\ln(1+h)} - e^{nh}\right|.$$

We now perform the two limits $h \to 0$, $n \to +\infty$ but with $nh = t$ finite and we obtain

$$e_n = \left|e^{n\left[h-h^2/2+O(h^3)\right]} - e^{nh}\right| = e^{nh}\left|e^{-nh^2/2+nO(h^3)} - 1\right| = e^t\left|e^{-th/2+tO(h^2)} - 1\right| = e^t\left[\frac{th}{2} + tO(h^2)\right]$$
$$= \frac{1}{2}te^t\, h \;+\; \text{higher-order terms.}$$

This implies that $e_n = O(h)$ as mentioned above.

From the above definitions, we can deduce some properties of the Forward Euler method.

Convergence of the Forward Euler method
The Forward Euler method has a global error $O(h)$: it is of **order 1** and it is therefore **convergent**.

Being of order 1, to increase by one digit the accuracy of the Forward Euler method (*i.e.*, to divide by a factor of 10 the global truncation error), you have to divide h by a factor of 10: you have to make 10 times more steps and your computation time is also multiplied by a factor of 10.

For the majority of numerical integrators of ODEs (see though a counter-example below), if the local truncation error is $O(h^{p+1})$, the global error is $O(h^p)$. Therefore, looking at the local truncation error is most of the time enough to decide whether the method is convergent.

4. Stability and stiffness

In the previous section, we have focused on the convergence of a numerical integrator, namely, with what happens when $h \to 0$. In practice, to reduce the computation time, you want to increase the value of h. Therefore, a natural question which emerges is how large you can make the step size to get a result that is still a correct estimate of the exact solution. For most methods, you cannot take h too large because numerical integrators of ODEs can become **unstable** (the result of the integration diverges from the exact solution). The theory of the **stability** of numerical integrators brings answers to the above question. However, it is relatively complex. Therefore, we will not delve into the details of such a theory, but instead illustrate stability issues when numerically integrating an ODE in the example below.

Consider the Cauchy problem

$$\frac{\mathrm{d}x}{\mathrm{d}t} = -ax, \quad x(0) = 1,$$

on $[0,\, 1]$ with $a > 0$. The exact solution is $x_{\mathrm{e}}(t) = e^{-at}$, while the Forward Euler method leads to

$$x_{n+1}^{(h)} = x_n^{(h)} - ahx_n^{(h)} = x_n^{(h)}(1-ah).$$

This is a geometric progression and for any n we have $x_n^{(h)} = (1-ah)^n$. The exact solution of the ODE $x_{\mathrm{e}}(t) = e^{-at}$

decays exponentially, and $x_n^{(h)}$ should be a decaying sequence with n. However, this is only the case if $-1 < 1-ah < 1$, otherwise the sequence diverges. The two behaviors are shown Fig. 2. Therefore, the integrator is stable if $h < 2/a$.

For certain ODEs, which are called **stiff**, you need to consider very small values of h in order for the method to be stable.

Stiff ODEs
Some ODEs are **stiff**, meaning that they may require a very small time step h in order for the numerical integrator to be stable. This often occurs when you have multiple characteristic timescales with different orders of magnitude in the differential equation.

The value of the step size h to recover stability depends on the ODE and also on the numerical solver.

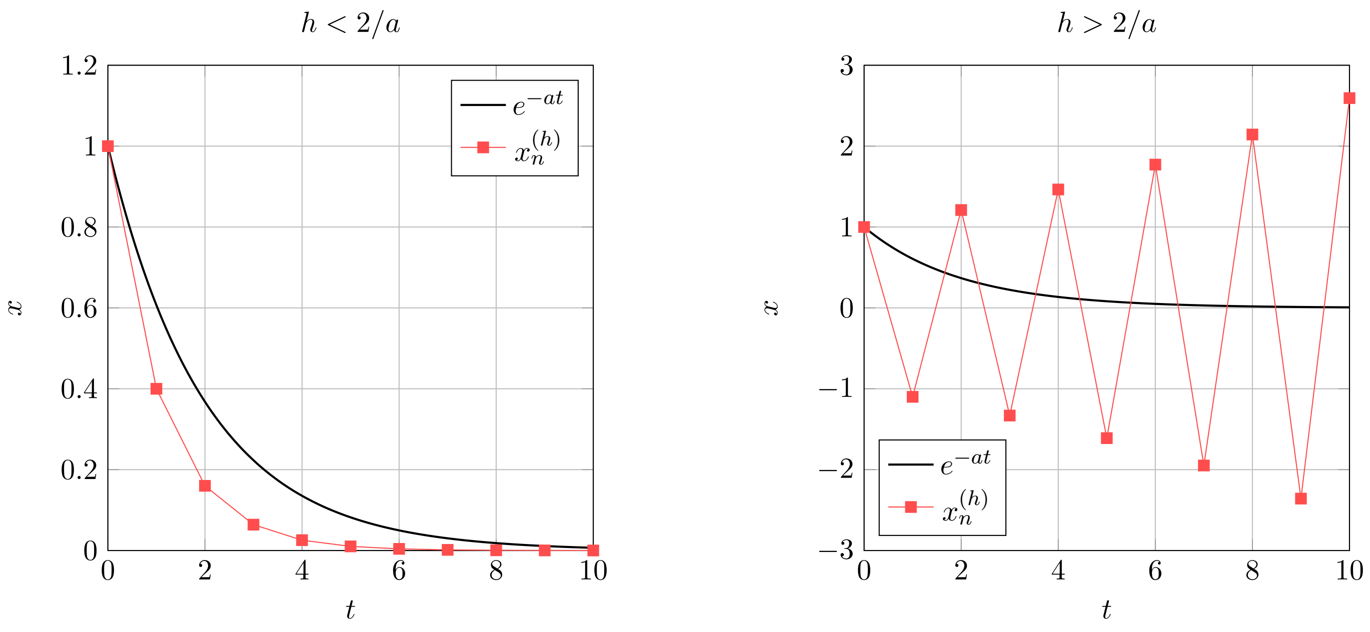

Figure 2: **Stability issues of the Forward Euler method.** We show the exact solution of the ODE $\mathrm{d}x/\mathrm{d}t = -ax$ for $a > 0$ and $x(0) = 1$ along with the numerical solution $x_n^{(h)}$ obtained from the Forward Euler method. In the left panel, the step size is $h < 2/a$ so that the integration scheme is stable. In the right panel, the step size is $h > 2/a$ and the integration scheme is unstable: the numerical solution diverges.

5. Conclusion

We can summarize the advantages and drawbacks of the Forward Euler method.

Advantages and Drawbacks of the Forward Euler method
Advantages:

- The Forward Euler method is **fast** (few operations at each time step).
- The Forward Euler method is **simple to implement** and only requires to store the value at the previous step.

Drawbacks:

- The Forward Euler method is **not very precise**.
- The Forward Euler method can be **unstable**.

III. The Backward Euler method

In this section, we discuss the Backward Euler method introduced above.

1. Definition

The Backward Euler method is defined as follows

Definition of the Backward Euler method
For the IVP given by Eq. (1), the Backward Euler method is a numerical integrator of ODEs given by the following recurrence relation:

$$\overrightarrow{x_{n+1}}^{(h)} = \overrightarrow{x_n} + hf(\overrightarrow{x_{n+1}}, t_{n+1}), \tag{13}$$

for $t_n = t_\mathrm{i} + nh$ and h the step size.

The Backward Euler method is *a priori* **implicit** because $\overrightarrow{x_{n+1}}^{(h)}$ also appears in the right-hand side of Eq. (13). There are three strategies to implement the algorithm.

First strategy: making the Backward Euler method explicit. In some cases, you can manipulate Eq. (13) analytically in order to obtain an explicit expression of $\overrightarrow{x_{n+1}}^{(h)} = g(\overrightarrow{x_n}^{(h)}, t_{n+1})$, see the example below. Once you have found an explicit expression, you can implement it in a similar way as the Forward Euler method.

Algorithm for the Backward Euler method when made explicit

```
$x \leftarrow x_\mathrm{i}$                ▷ Initial condition.
$t \leftarrow t_\mathrm{i}$                ▷ Initial time.
STORE($x$)                 ▷ Store the initial value of $x$.
STORE($t$)                 ▷ Store the initial value of $t$.
while $t < t_\mathrm{f}$ do
    $t \leftarrow t + h$                ▷ Update the time.
    $x \leftarrow x + hg(x, t)$          ▷ Update the solution.
    STORE($x$)             ▷ Store the current value of $x$.
    STORE($t$)             ▷ Store the current value of $t$.
```

We present an example where we can turn the implicit equation defining the Backward Euler method into an explicit expression. Consider the Cauchy problem already introduced in the example above

$$\frac{\mathrm{d}x}{\mathrm{d}t} = -ax, \quad x(0) = 1,$$

on $[0,\, 1]$ with $a > 0$. The Backward Euler method is defined by the equation $x_{n+1}^{(h)} = x_n^{(h)} - ahx_{n+1}^{(h)}$, that you can transform into an explicit equation for $x_{n+1}^{(h)}$:

$$x_{n+1}^{(h)} = \frac{x_n^{(h)}}{1 + ah}.$$

Of course, it is not always possible to make the Backward Euler method explicit. We thus propose two other strategies.

Second strategy: finding the solution by iteration. Equation (13) implies that $\overrightarrow{x_{n+1}}^{(h)}$ is a fixed point of the function $F : \vec{x} \mapsto \overrightarrow{x_n}^{(h)} + hf(\vec{x}, t_{n+1})$. There is a mathematical theorem which tells you that, under reasonable properties of the function F, if you consider the sequence $(\overrightarrow{y_p})_{p\in\mathbb{N}}$ defined by the recursion relation $\overrightarrow{y_{p+1}} = F(\overrightarrow{y_p})$, this sequence converges to the fixed point of F, which is precisely the solution $\overrightarrow{x_{n+1}}^{(h)}$ of the Backward Euler equation. As a consequence, the idea is to iterate the function F, starting from $\overrightarrow{y_0} = \overrightarrow{x_n}^{(h)}$ until the sequence $(\overrightarrow{y_p})_{p\in\mathbb{N}}$ converges. In practice, you have to stop the iteration when the relative change between two iterations is smaller than a given threshold ϵ.

Algorithm for the Backward Euler method with an iteration scheme

```
$x \leftarrow x_\mathrm{i}$                                   ▷ Initial condition.
$t \leftarrow t_\mathrm{i}$                                   ▷ Initial time.
STORE($x$)                                    ▷ Store the initial value of $x$.
STORE($t$)                                    ▷ Store the initial value of $t$.
while $t < t_\mathrm{f}$ do
    $t \leftarrow t + h$                              ▷ Update the time.
    $y \leftarrow x$                                  ▷ Initiate the sequence for the research of the fixed point.
    $z \leftarrow +\infty$ ▷ $z$ represents the value of $y$ at the previous iteration: this value is stored to check whether $y$ converges to the fixed point.
    while $\|y - z\| > \epsilon \|y\|$ do ▷ Iterate the function $F$ until the sequence converges to the fixed point up to a given threshold $\epsilon$.
        $z \leftarrow y$
        $y \leftarrow x + hf(y, t)$
    $x \leftarrow y$                                  ▷ Update the value of the solution.
    STORE($x$)                                ▷ Store the current value of $x$.
    STORE($t$)                                ▷ Store the current value of $t$.
```

However, this iterative method can be unstable: the success of the iterative process depends on the mathematical properties of F, and therefore on the value of the step size h which should be small enough. We illustrate this in the example below, and we provide next a last strategy to solve Eq. (13).

We come back to the Cauchy problem considered above

$$\frac{\mathrm{d}x}{\mathrm{d}t} = -ax, \quad x(0) = 1,$$

on $[0,\, 1]$ with $a > 0$, and we apply the iteration scheme at step $n+1$. We thus define the sequence $(y_p)_{p\in\mathbb{N}}$ such that $y_0 = x_n^{(h)}$ and $y_{p+1} = x_n^{(h)} - ahy_p$. You can thus obtain a generic expression for y_p (this is a good mathematical exercise):

$$y_p = \frac{x_n^{(h)}}{1+ah}\left[1 - (-ah)^{p+1}\right].$$

You then see that $(y_p)_{p\in\mathbb{N}}$ converges to the fixed point $x_n^{(h)}/(1+ah)$ (see the previous example where we made the implicit Euler method explicit) only if $-1 < -ah < 1$, *i.e.*, $h < 1/a$. We illustrate the two behaviors $h < 1/a$ and $h > 1/a$ in Fig. 3. You can note that this criterion is roughly similar to the stability criterion for the Forward Euler method (we found $h < 2/a$ in that case).

Third strategy: finding the solution with a root-finding algorithm. Equation (13) implies that $\overrightarrow{x_{n+1}}^{(h)}$ is a root of the function $G : \vec{x} \mapsto \vec{x} - \overrightarrow{x_n}^{(h)} - hf(\vec{x}, t_{n+1})$, namely, $G(\overrightarrow{x_{n+1}}^{(h)}) = \vec{0}$. There is a well-known method to find the root of a function, which is called the Newton method, that we detail now. The idea is again to build a sequence $(\overrightarrow{y_p})_{p\in\mathbb{N}}$ with $\overrightarrow{y_0} = \overrightarrow{x_n}^{(h)}$, and to iterate such that $(\overrightarrow{y_p})_{p\in\mathbb{N}}$ converges to the root of G. The iteration is defined as follows: $\overrightarrow{y_{p+1}} = \overrightarrow{y_p} - \overrightarrow{\delta_p}$ where $\overrightarrow{\delta_p} = (\delta_{p,1},\, \dots,\, \delta_{p,d}) \in \mathbb{R}^d$ is the solution of the linear system

$$\begin{pmatrix} \frac{\partial G_1}{\partial y_1}(\overrightarrow{y_p}) & \dots & \frac{\partial G_1}{\partial y_d}(\overrightarrow{y_p}) \\ \vdots & & \vdots \\ \frac{\partial G_d}{\partial y_1}(\overrightarrow{y_p}) & \dots & \frac{\partial G_d}{\partial y_d}(\overrightarrow{y_p}) \end{pmatrix} \overrightarrow{\delta_p} = G(\overrightarrow{y_p}), \tag{14}$$

with $G(\vec{y}) = (G_1(\vec{y}), \dots, G_d(\vec{y}))$. The matrix of the partial derivatives in the left hand-side is called the Jacobian matrix. This method is implemented in Python, look at the documentation page of `scipy.optimize`. As for the iteration procedure, you have to truncate the sequence $(\overrightarrow{y_p})_{p\in\mathbb{N}}$ when the relative change between two iterations is smaller than a given threshold ϵ. We illustrate the method with an example below, provide a mathematical proof in a remark, and finally give a schematic algorithm.

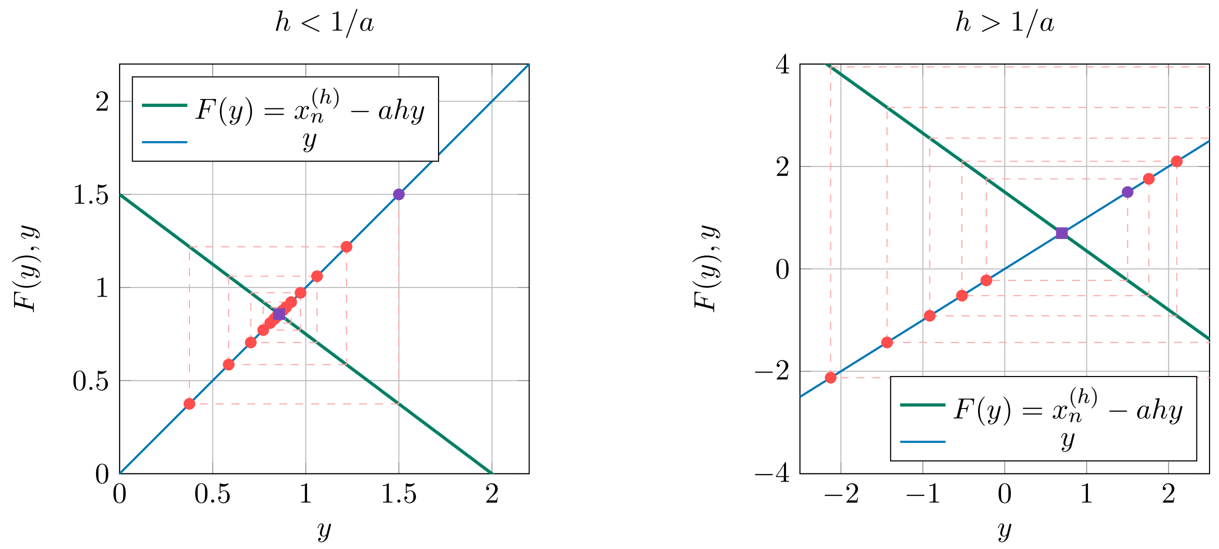

Figure 3: **Convergence issue when solving the Backward Euler equation with an iteration scheme.** We illustrate the method to solve for step $n+1$ the ODE $\mathrm{d}x/\mathrm{d}t = -ax$ for $a > 0$ from step n using the Backward Euler method with an iteration scheme. We define the sequence $y_0 = x_n^{(h)}$ (violet disks) and $y_{p+1} = F(y_p)$ with $F(y) = x_n^{(h)} - ahy$. For a step size h small enough ($h < 1/a$), the iteration scheme converges to the fixed point (violet square). Instead for a too large step size h ($h > 1/a$), the iteration scheme diverges and is unstable.

We come back to the Cauchy problem considered above

$$\frac{\mathrm{d}x}{\mathrm{d}t} = -ax, \quad x(0) = 1,$$

on $[0,\, 1]$ with $a > 0$, and we apply the root-finding scheme at step $n+1$. We thus define the sequence $(y_p)_{p\in\mathbb{N}}$ such that $y_0 = x_n^{(h)}$ and $y_{p+1} = y_p - \delta_p$ with

$$\delta_p = \frac{G(y_p)}{G'(y_p)}, \quad \text{with} \quad G(y) = y - x_n^{(h)} + ahy.$$

You then easily get that $\delta_p = y_p - x_n^{(h)}/(1+ah)$ so that $y_{p+1} = x_n^{(h)}/(1+ah)$. As a result, the sequence always converges to $x_{n+1}^{(h)}$ (see the above example where we made the implicit Backward Euler method explicit) in one step, whatever the value of h.

In this remark, we prove the Newton method. Imagine that $\overrightarrow{y_p}$ is close to $\overrightarrow{x_{n+1}}^{(h)}$. The difference $\overrightarrow{\delta_p} = \overrightarrow{y_p} - \overrightarrow{x_{n+1}}^{(h)}$ is thus much smaller than $\overrightarrow{y_p}$: $\|\overrightarrow{\delta_p}\| \ll \|\overrightarrow{y_p}\|$. If we use the fact that $\overrightarrow{x_{n+1}}^{(h)}$ is a root of G, it means that

$$\forall\, i \in [\![1,\, d]\!]\; G_i(\overrightarrow{x_{n+1}}^{(h)}) = 0.$$

If we now use the fact that $\overrightarrow{x_{n+1}}^{(h)} = \overrightarrow{y_p} - \overrightarrow{\delta_p}$, we have that

$$\forall\, i \in [\![1,\, d]\!]\; G_i(y_{p,1} - \delta_{p,1}, \ldots, y_{p,d} - \delta_{p,d}) = 0.$$

We can then do a Taylor expansion at first order, and we obtain that

$$\forall\, i \in [\![1,\, d]\!]\; G_i(\overrightarrow{y_p}) - \sum_{j=1}^{d} \frac{\partial G_i}{\partial y_j}(\overrightarrow{y_p})\delta_{p,j} = 0.$$

Putting this into a matrix form gives you Eq. (14).

Algorithm for the Backward Euler method with a root-finding scheme

```
$x \leftarrow x_\mathrm{i}$    ▷ Initial condition.
$t \leftarrow t_\mathrm{i}$    ▷ Initial time.
STORE($x$)    ▷ Store the initial value of $x$.
STORE($t$)    ▷ Store the initial value of $t$.
while $t < t_\mathrm{f}$ do
    $t \leftarrow t + h$    ▷ Update the time.
    $y \leftarrow x$    ▷ Initiate the sequence for the research of the root.
    $z \leftarrow +\infty$ ▷ $z$ represents the value of $y$ at the previous iteration: this value is stored to check whether $y$ converges to the root.
    while $\|y - z\| > \epsilon \|y\|$ do    ▷ Iterate until convergence to the root up to a given threshold $\epsilon$.
        $z \leftarrow y$
        $G_\mathrm{eval} \leftarrow \mathrm{G}(y)$    ▷ Evaluate $G(\overrightarrow{y_p})$.
        $J_\mathrm{eval} \leftarrow \mathrm{J}(y)$    ▷ Evaluate the Jacobian matrix $J(\overrightarrow{y_p})$ of the partial derivatives.
        $\delta \leftarrow$ SOLVE_LINEAR_SYSTEM($J_\mathrm{eval}, G_\mathrm{eval}$)    ▷ Solve the linear system defining $\overrightarrow{\delta_p}$.
        $y \leftarrow y - \delta$    ▷ Update the value of the sequence.
    $x \leftarrow y$    ▷ Update the value of the solution.
    STORE($x$)    ▷ Store the current value of $x$.
    STORE($t$)    ▷ Store the current value of $t$.
```

We discuss the properties of the Backward Euler method in the next section.

2. Properties

We give the main properties of the Backward Euler method, which are the consequences of the Taylor expansion given by Eq. (3).

Consistency and convergence of the Backward Euler method
The Backward Euler method has a local truncation error $O(h^2)$: it is **consistent**.

The Backward Euler method has a global truncation error $O(h)$ (method of order 1): it is **convergent**.

Its characteristics are thus similar to the Forward Euler method. However, it may require much more operations than the Forward Euler method if you use the strategy of the fixed point or the one of root-finding to solve Eq. (13). Still, the Backward Euler method has one advantage with respect to the Forward Euler method: it is **more stable** if you can make the Backward Euler method explicit, or if you use a root-finding algorithm. We illustrate this in the example below.

We come back to the Cauchy problem

$$\frac{\mathrm{d}x}{\mathrm{d}t} = -ax, \quad x(0) = 1,$$

on $[0,\,1]$ with $a > 0$. We have seen above that the Backward Euler method can be made explicit:

$$x_{n+1}^{(h)} = \frac{x_n^{(h)}}{1 + ah}.$$

This implies that $x_n^{(h)}$ has a geometric progression, leading to

$$x_n^{(h)} = \frac{1}{(1 + ah)^n} = \left(\frac{1}{1 + ah}\right)^n.$$

For all values of h, this sequence decays with n, because $0 < 1/(1 + ah) < 1$, and the method is thus stable.

We have also discussed above the stability conditions of the iteration scheme and of the root-finding scheme.

We end this section by summarizing the advantages and drawbacks of the Backward Euler method.

Advantages and Drawbacks of the Backward Euler method
Advantages:

- The Backward Euler method is **relatively simple** to implement and only requires to store the value at the previous step.
- The Backward Euler method is **more stable** if you can make it explicit or if you use a root-finding algorithm: it is better for stiff problems.

Drawbacks:

- The Backward Euler method may be **slow** if you have to iterate in order to solve the equation defining the value of the solution at the next step.
- The Backward Euler method is **not very precise**.

IV. The Trapezoidal method

In this section, we briefly discuss the Trapezoidal method introduced above.

1. Definition

The Trapezoidal method is defined as follows

Definition of the Trapezoidal method
For the IVP given by Eq. (1), the Trapezoidal method is a numerical integrator of ODEs given by the following recurrence relation:

$$\overrightarrow{x_{n+1}}^{(h)} = \overrightarrow{x_n} + \frac{h}{2}\left[f(\overrightarrow{x_n}, t_n) + f(\overrightarrow{x_{n+1}}, t_{n+1})\right], \tag{15}$$

for $t_n = t_\mathrm{i} + nh$ and h the step size.

The Trapezoidal method is *a priori* **implicit** because $x_{n+1}^{(h)}$ also appears in the right-hand side of Eq. (15). There are four strategies to implement the algorithm.

First strategy: making the Trapezoidal method explicit. As for the Backward Euler method, you can manipulate Eq. (15) analytically in order to obtain an explicit expression of $\overrightarrow{x_{n+1}}^{(h)} = g(\overrightarrow{x_n}^{(h)}, t_n, t_{n+1})$, see the example below. Once you have found an explicit expression, you can implement it in a similar way as the Forward Euler method.

We present an example where we can turn the implicit equation defining the Trapezoidal method into an explicit expression. Consider again the Cauchy problem

$$\frac{\mathrm{d}x}{\mathrm{d}t} = -ax, \quad x(0) = 1,$$

on $[0,\,1]$ with $a > 0$. The Trapezoidal method is defined by the equation

$$x_{n+1}^{(h)} = x_n^{(h)} - \frac{ah}{2}\left[x_n^{(h)} + x_{n+1}^{(h)}\right],$$

that you can transform into an explicit equation for $x_{n+1}^{(h)}$:

$$x_{n+1}^{(h)} = x_n^{(h)}\frac{1 - ah/2}{1 + ah/2}.$$

Of course, as before, this is not always possible to make the Trapezoidal method explicit. In the latter case, you have to use the methods which have been introduced in the previous section about the Backward Euler method (**second strategy: finding the solution by iteration** and **third strategy: finding the solution with a root-finding algorithm**). Or you can use a much simpler method which is specific to the Trapezoidal method.

Fourth strategy: using a predictor-corrector strategy. The idea is to make a first guess of $\overrightarrow{x_{n+1}}^{(h)}$ (predictor $\vec{y}$) using a Forward Euler method, and then to use this predictor value to compute the actual value of $\overrightarrow{x_{n+1}}^{(h)}$ (corrector) by replacing $\overrightarrow{x_{n+1}}^{(h)}$ by $\vec{y}$ in the right-hand side of Eq. (15). In other words, you have turned an implicit one-stage method into a two-stage explicit method. The combination of the Trapezoidal method with a predictor-corrector strategy is sometimes called the Heun's method.

Algorithm for the Trapezoidal method with the predictor-corrector scheme

```
$x \leftarrow x_i$                                          ▷ Initial condition.
$t \leftarrow t_i$                                          ▷ Initial time.
STORE($x$)                                                  ▷ Store the initial value of $x$.
STORE($t$)                                                  ▷ Store the initial value of $t$.
while $t < t_f$ do
    $t' \leftarrow t$                                       ▷ Store the time at step $n$.
    $t \leftarrow t + h$                                    ▷ Update the time at step $n+1$.
    $y \leftarrow x + hf(x, t')$                            ▷ Compute the predictor.
    $x \leftarrow x + (h/2)f(x, t') + (h/2)f(y, t)$         ▷ Compute the corrector.
    STORE($x$)                                              ▷ Store the current value of $x$.
    STORE($t$)                                              ▷ Store the current value of $t$.
```

Another way of seeing the predictor-corrector strategy is to notice that it is equivalent to the iteration scheme (second strategy) but truncated after the first iteration. As a consequence, this method can also be unstable for stiff problems, as illustrated in the example below.

We come back to the Cauchy problem considered above

$$\frac{\mathrm{d}x}{\mathrm{d}t} = -ax, \quad x(0) = 1,$$

on $[0,\ 1]$ with $a > 0$, and we apply the predictor-corrector scheme at step $n+1$. We thus have $y = x_n^{(h)} - ahx_n^{(h)}$ and then

$$x_{n+1}^{(h)} = x_n^{(h)} - \frac{ah}{2}x_n^{(h)} - \frac{ah}{2}y,$$

leading to

$$x_{n+1}^{(h)} = x_n^{(h)}\left[1 - ah + \frac{(ah)^2}{2}\right].$$

You can recognize the Taylor expansion of $t \mapsto e^{-at}$ (the exact solution) up to second order. The estimates $x_n^{(h)}$ follow a geometric progression and we can eventually obtain a closed expression for $x_n^{(h)}$:

$$x_n^{(h)} = \left[1 - ah + \frac{(ah)^2}{2}\right]^n.$$

The exact solution is an exponentially decaying function, and $x_n^{(h)}$ should be a decaying sequence with n. However, this is only the case if $-1 < 1 - ah + (ah)^2/2 < 1$, resulting in $h < 2/a$ in order for the method to be stable. Otherwise, the sequence diverges. This is the same stability criterion as for the Forward Euler method.

We discuss the properties of the Trapezoidal method in the next section.

2. Properties

We give the main properties of the Trapezoidal method, which are the consequences of the Taylor expansions given by Eq. (4) and Eq. (5).

Consistency and convergence of the Trapezoidal method
The Trapezoidal method has a local truncation error $O(h^3)$: it is **consistent**.

The Trapezoidal method has a global truncation error $O(h^2)$ (method of order 2): it is **convergent**. Said differently, to increase by one digit the accuracy of the method (*i.e.*, to divide by a factor of 10 the global truncation error), you have to divide h by a factor of $\sqrt{10} \simeq 3$ only: you have to make 3 times more steps and your computation time is also multiplied by a factor of 3.

We end this section by summarizing the advantages and drawbacks of the Trapezoidal method.

Advantages and Drawbacks of the Trapezoidal method
Advantages:

- The Trapezoidal method is **relatively simple** to implement and only requires to store the value at the previous step.
- The Trapezoidal method is **more stable** if you can make it explicit or if you use the Newton method: it is better for stiff problems.
- The Trapezoidal method is **more precise** than the Euler methods (see also Fig. 1).

Drawbacks:

- The Trapezoidal method may be **slow** if you have to iterate in order to solve the equation defining the value of the solution at the next step.

V. Runge-Kutta methods

We now discuss **Runge-Kutta methods** which are heavily used in Physics for their high degree of precision and their easy implementation.

1. Definition

In the previous sections, we have discussed the Euler methods and the Trapezoidal method. The latter is more precise than the formers. The change in precision comes from a different choice of points from which the Taylor expansions in Eqs. (2), (3), (4) and (5) are performed: the Taylor expansion is done from $t = t_n$ for the Forward Euler method, $t = t_{n+1}$ for the Backward Euler method, and $t = t_n + h/2$ for the Trapezoidal method. To derive Runge-Kutta methods, you just have to consider several points between t_n and t_{n+1} and play with the Taylor expansions to cancel as many powers of h as you want in order to increase the accuracy. You can construct an entire family of Runge-Kutta methods, see J. C. Butcher, *Numerical Methods for Ordinary Differential Equations*. In these notes, we only discuss the most famous Runge-Kutta method: RK4.

Definition of the RK4 method
For the IVP given by Eq. (1), the RK4 method is a numerical integrator of ODEs given by the following recurrence relations:

$$\begin{cases} \overrightarrow{k_1} = hf(\overrightarrow{x_n}, t_n), \\ \overrightarrow{k_2} = hf(\overrightarrow{x_n} + \overrightarrow{k_1}/2, t_n + h/2), \\ \overrightarrow{k_3} = hf(\overrightarrow{x_n} + \overrightarrow{k_2}/2, t_n + h/2), \\ \overrightarrow{k_4} = hf(\overrightarrow{x_n} + \overrightarrow{k_3}, t_n + h), \\ \overrightarrow{x_{n+1}}^{(h)} = \overrightarrow{x_n}^{(h)} + \frac{1}{6}\left(\overrightarrow{k_1} + 2\overrightarrow{k_2} + 2\overrightarrow{k_3} + \overrightarrow{k_4}\right), \end{cases} \tag{16}$$

for $t_n = t_i + nh$ and h the step size.

The RK4 method is **explicit**. It is a one-step method (because you only require the knowledge of the solution at the previous step) but it has multiple stages. We can propose a simple algorithmic representation. Note however that this method is already implemented in Python, see the documentation.

Algorithm for the RK4 method

```
$x \leftarrow x_\mathrm{i}$                                    ▷ Initial condition.
$t \leftarrow t_\mathrm{i}$                                    ▷ Initial time.
STORE($x$)                                   ▷ Store the initial value of $x$.
STORE($t$)                                   ▷ Store the initial value of $t$.
while $t < t_\mathrm{f}$ do
    $k_1 \leftarrow hf(x, t)$
    $k_2 \leftarrow hf(x + k_1/2, t + h/2)$
    $k_3 \leftarrow hf(x + k_2/2, t + h/2)$
    $k_4 \leftarrow hf(x + k_3, t + h)$
    $x \leftarrow x + (k_1 + 2k_2 + 2k_3 + k_4)/6$           ▷ Update the solution.
    $t \leftarrow t + h$                                ▷ Update the time.
    STORE($x$)                               ▷ Store the current value of $x$.
    STORE($t$)                               ▷ Store the current value of $t$.
```

We discuss the properties of the RK4 method in the next section.

2. Properties

We give the main properties of the RK4 method, which are again the consequences of the Taylor expansions which define it.

Consistency and convergence of the RK4 method
The RK4 method has a local truncation error $O(h^5)$: it is **consistent**.

The RK4 method has a global truncation error $O(h^4)$ (method of order 4): it is **convergent**. Said differently, to increase by one digit the accuracy of the method (*i.e.*, to divide by a factor of 10 the global truncation error), you have to divide h by a factor of $\sqrt[4]{10} \simeq 1.8$ only: you have to make less than twice more steps and your computation time is also multiplied by a factor of 1.8 roughly.

Although the RK4 method is very precise, it can be unstable when trying to solve a stiff problem, as exemplified below.

Consider the Cauchy problem

$$\frac{\mathrm{d}x}{\mathrm{d}t} = -ax, \quad x(0) = 1,$$

on $[0,\, 1]$ with $a > 0$. The RK4 method applied to this problem gives:

$$\begin{cases} k_1 = -ahx_n^{(h)}, \\ k_2 = -ah\left(1 - \dfrac{ah}{2}\right)x_n^{(h)}, \\ k_3 = -ah\left[1 - \dfrac{ah}{2}\left(1 - \dfrac{ah}{2}\right)\right]x_n^{(h)}, \\ k_4 = -ah\left\{1 - ah\left[1 - \dfrac{ah}{2}\left(1 - \dfrac{ah}{2}\right)\right]\right\}x_n^{(h)}, \end{cases}$$

such that you eventually get that

$$x_{n+1}^{(h)} = x_n^{(h)}\left[1 - ah + \frac{(ah)^2}{2} - \frac{(ah)^3}{6} + \frac{(ah)^4}{24}\right].$$

In the brackets, you recognize the Taylor expansion of $t \mapsto e^{-at}$ (which is the exact solution) up to fourth order. This is a geometric progression and for any n we have

$$x_n^{(h)} = \left[1 - ah + \frac{(ah)^2}{2} - \frac{(ah)^3}{6} + \frac{(ah)^4}{24}\right]^n.$$

The exact solution is an exponentially decaying function, and $x_n^{(h)}$ should be a decaying sequence with n. However, this is only the case if the common ratio (the quantity into brackets) lies between -1 and 1. You can prove that it amounts to

$$ah < \frac{1}{3}\left[4 - 10\left(\frac{2}{43+9\sqrt{29}}\right)^{1/3} + 2^{2/3}\left(43+9\sqrt{29}\right)^{1/3}\right] \simeq 2.78529...$$

Therefore, the RK4 method is stable if the above inequality is satisfied. Otherwise, the sequence diverges and the RK4 method is unstable. You can note that the stability criterion derived above is less restrictive than for the Forward Euler method (it was $ah < 2$ in that case).

We end this section by summarizing the advantages and drawbacks of the RK4 method.

Advantages and Drawbacks of the RK4 method

Advantages:

- The RK4 method is **relatively simple** to implement and only requires to store the value at the previous step.
- The RK4 method is **relatively fast** (few operations at each time step).
- The RK4 method is **much more precise** than the Euler methods or the Trapezoidal method.

Drawbacks:

- The RK4 method can be **unstable** when you have to solve stiff problems. However, it is more stable than the Forward Euler method.

To cure the stability issues of the RK4 methods, implicit Runge-Kutta methods have been introduced, but they are not discussed in these lecture notes.

VI. Higher-order Ordinary Differential Equations

So far, we have only discussed Initial Value Problems (IVPs) with first-order differential equations. We now discuss how to deal with higher-order differential equations in general. In the next section, we consider particular second-order differential equations for which specific methods have been introduced.

Any ODE of order $q > 1$ can be recast into a system of q first-order ODEs by introducing auxiliary variable. Imagine that you want to solve a scalar ODE of the form

$$\frac{\mathrm{d}^q x}{\mathrm{d}t^q} = f\left(x, \frac{\mathrm{d}x}{\mathrm{d}t}, \dots, \frac{\mathrm{d}^{q-1}x}{\mathrm{d}t^{q-1}}, t\right),$$

where $x \in \mathbb{R}$. Then, you can define $x_0 = x$, $x_1 = \mathrm{d}x/\mathrm{d}t$, ..., $x_{q-1} = \mathrm{d}^{q-1}x/\mathrm{d}t^{q-1}$ such that the scalar ODE of order q can be written as a system of q coupled first-order ODEs

$$\begin{cases} \dfrac{\mathrm{d}x_0}{\mathrm{d}t} & = x_1, \\ \dfrac{\mathrm{d}x_1}{\mathrm{d}t} & = x_2, \\ & \vdots \\ \dfrac{\mathrm{d}x_{q-2}}{\mathrm{d}t} & = x_{q-1}, \\ \dfrac{\mathrm{d}x_{q-1}}{\mathrm{d}t} & = f(x_0, x_1, \dots, x_{q-1}, t) \end{cases},$$

that you can eventually recast in a vectorial form:

$$\frac{\mathrm{d}\vec{x}}{\mathrm{d}t} = F(\vec{x},t), \text{ with } \vec{x} = (x_0,\, x_1,\, \dots,\, x_{q-1}) \text{ and } F(\vec{x},t) = (x_1,\, x_2,\, \dots, x_{q-1},\, f(x_0,\, x_1,\, \dots,\, x_{q-1},\, t)).$$

You can then apply any of the methods introduced above to deal with first-order ODEs.

VII. The Verlet algorithm for second-order Ordinary Differential Equations

In this section, we discuss the numerical integration of Cauchy problems of the form:

$$\begin{cases} \dfrac{\mathrm{d}^2\vec{x}}{\mathrm{d}t^2} = f(\vec{x}), \\ \vec{x}(t_\mathrm{i}) = \vec{x}_\mathrm{i}, \\ \dfrac{\mathrm{d}\vec{x}}{\mathrm{d}t}(t_\mathrm{i}) = \vec{v}_\mathrm{i}, \end{cases} \tag{17}$$

where $\vec{x} \in \mathbb{R}^d$ is a d-dimensional vector, and where $f : \mathbb{R}^d \to \mathbb{R}^d$. Note that this Cauchy problem is not the most general one for second-order differential equations for two reasons:

- the function in the right-hand side **only depends on $\vec{x}$ and not on its first derivative**;
- the function in the right-hand side **does not depend explicitely on time**: the system is autonomous.

Despite these specificities, being able to solve such equations is of paramount importance because they naturally emerge in Physics when solving Newton's equations for the **mechanics of conservative systems**.
We can rewrite the above Cauchy problem as a system of vectorial first-order ODEs by introducing the auxiliary variable $\vec{v} = \mathrm{d}\vec{x}/\mathrm{d}t$ (the velocity):

$$\begin{cases} \dfrac{\mathrm{d}\vec{x}}{\mathrm{d}t} = \vec{v}, \\ \dfrac{\mathrm{d}\vec{v}}{\mathrm{d}t} = f(\vec{x}), \\ \vec{x}(t_\mathrm{i}) = \vec{x}_\mathrm{i}, \\ \vec{v}(t_\mathrm{i}) = \vec{v}_\mathrm{i}, \end{cases} \tag{18}$$

1. Definition

We now define the **velocity Verlet algorithm** to solve the above Cauchy problem. For more details, you can read M. P. Allen and D. J. Tildesley, *Computer Simulations of Liquids*.

Definition of the velocity Verlet algorithm
For the IVP given by Eq. (17) [or Eq. (18)], the velocity Verlet algorithm is a numerical integrator given by the following recurrence relations:

$$\begin{cases} \overrightarrow{x_{n+1}}^{(h)} = \overrightarrow{x_n}^{(h)} + h\overrightarrow{v_n}^{(h)} + \dfrac{h^2}{2} f(\overrightarrow{x_n}^{(h)}), \\ \overrightarrow{v_{n+1}}^{(h)} = \overrightarrow{v_n}^{(h)} + \dfrac{h}{2}\left[f(\overrightarrow{x_n}^{(h)}) + f(\overrightarrow{x_{n+1}}^{(h)})\right], \end{cases} \tag{19}$$

for $t_n = t_\mathrm{i} + nh$ and h the step size.

The velocity Verlet method is **explicit**. We can thus propose a simple algorithmic representation.

Algorithm for the velocity Verlet method

```
$x \leftarrow x_\mathrm{i}$                              ▷ Initial position.
$v \leftarrow v_\mathrm{i}$                              ▷ Initial velocity.
$a \leftarrow f(x_\mathrm{i})$                           ▷ Initial acceleration.
$t \leftarrow t_\mathrm{i}$                              ▷ Initial time.
STORE($x$)                                 ▷ Store the initial value of position.
STORE($v$)                                 ▷ Store the initial value of velocity.
STORE($t$)                                 ▷ Store the initial value of time.
while $t < t_\mathrm{f}$ do
    $x \leftarrow x + hv + (h^2/2)a$                ▷ Update the position.
    $v \leftarrow v + (h/2)a$                       ▷ Update partially the velocity.
    $a \leftarrow f(x)$                             ▷ Update the acceleration.
    $v \leftarrow v + (h/2)a$                       ▷ Update the velocity.
    $t \leftarrow t + h$                            ▷ Update the time.
    STORE($x$)                             ▷ Store the current value of position.
    STORE($v$)                             ▷ Store the current value of velocity.
    STORE($t$)                             ▷ Store the current value of time.
```

We discuss the properties of the velocity Verlet method in the next section.

2. Properties

Consistency and convergence of the velocity Verlet method

The velocity Verlet method has a local truncation error $O(h^4)$: it is **consistent**.

The velocity Verlet method has a global truncation error $O(h^2)$ (method of order 2): it is **convergent**. Said differently, to increase by one digit the accuracy of the method (*i.e.*, to divide by a factor of 10 the global truncation error), you have to divide h by a factor of $\sqrt{10} \simeq 3$ only.

The RK4 method is more precise than the velocity Verlet method. So why should we prefer the velocity Verlet method when integrating Cauchy problems of the form given by Eq. (17)? We give the main motivation below and illustrate it with an example after.

Symplecticness of the velocity Verlet algorithm

The velocity Verlet algorithm is **symplectic**. In particular, for conservative systems for which the total energy is constant, the algorithm approximately preserves this conservation law.

Instead, the Runge-Kutta 4 method is **not symplectic**. For conservative systems, the total energy systematically decreases.

We consider the Cauchy problem corresponding to a harmonic oscillator:

$$\begin{cases} \dfrac{\mathrm{d}^2x}{\mathrm{d}t^2} = -\omega^2 x, \\ x(0) = 1, \\ \dfrac{\mathrm{d}x}{\mathrm{d}t}(0) = 0. \end{cases}$$

We integrate it with the same step size h using the velocity Verlet algorithm and the Runge-Kutta 4 method. The results are presented Fig. 4.

We first analyze the numerical solution obtained in a small time interval, and we confront it with the exact solution $\cos(\omega t)$ (see the left panels). Both methods agree well with the exact solution, although the numerical estimates from the velocity Verlet algorithm start to deviate from the exact solution at the end of the time window. This comes from the fact that the Runge-Kutta 4 method has a better global truncation error than the velocity Verlet algorithm.

We now analyze the numerical solution obtained in a larger time interval. The harmonic oscillator is a conservative system, meaning that its total energy

$$E = \frac{1}{2}m\left(\frac{\mathrm{d}x}{\mathrm{d}t}\right)^2 + \frac{1}{2}m\omega^2 x^2$$

is constant during the dynamics. We now assess if this property is preserved by the velocity Verlet algorithm and the Runge-Kutta 4 method. It turns out that for the velocity Verlet algorithm, the energy fluctuates around a value which is close to the exact value of the energy. In other words, energy conservation is approximately preserved by the velocity Verlet algorithm. Instead, for the Runge-Kutta 4 method, the energy systematically decreases from its exact value at time 0, meaning that energy conservation is not preserved by the Runge-Kutta 4 method.

We can therefore give the motivation to prefer the velocity Verlet algorithm.

Choice of the velocity Verlet algorithm
The velocity Verlet algorithm is more suitable than the Runge-Kutta 4 method to perform long simulations of systems with conserved quantities.

Although the velocity Verlet algorithm is symplectic, it can be unstable when trying to solve problems, as shown in the example below.

We come back to the Cauchy problem studied above:

$$\begin{cases} \dfrac{\mathrm{d}^2x}{\mathrm{d}t^2} = -\omega^2 x, \\ x(0) = 1, \\ \dfrac{\mathrm{d}x}{\mathrm{d}t}(0) = 0. \end{cases}$$

If you solve it using the velocity Verlet algorithm, you get the two coupled equations (with $v = \mathrm{d}x/\mathrm{d}t$):

$$\begin{cases} x_{n+1}^{(h)} = x_n^{(h)}\left(1 - \dfrac{\omega^2h^2}{2}\right) + hv_n^{(h)}, \\ v_{n+1}^{(h)} = v_n^{(h)}\left(1 - \dfrac{\omega^2h^2}{2}\right) - h\omega^2 x_n^{(h)}\left(1 - \dfrac{\omega^2h^2}{4}\right), \end{cases}$$

that you can recast into a simple second-order recursion relation for $x_n^{(h)}$:

$$x_{n+2}^{(h)} = (2 - \omega^2h^2)x_{n+1}^{(h)} - x_n^{(h)}.$$

The solution of this recursion relation is of the form $x_n^{(h)} = Ar_1^n + Br_2^n$, with r_1 and r_2 the two roots of the quadratic equation

$$r^2 - (2 - \omega^2h^2)r + 1 = 0.$$

If the discriminant $\Delta = (2-\omega^2h^2)^2 - 4 < 0$ then the two roots are complex conjugate. This happens if $\omega h < 2$. In this case, as $r_1r_2 = 1$ (relation between the product of the two roots and the coefficients of the quadratic equation), they verify $|r_1| = |r_2| = 1$. You can thus write them as $r_1 = e^{\mathrm{i}\phi}$, $r_2 = e^{-\mathrm{i}\phi}$. If you finally inject this form into the quadratic equation, you get

$$\cos\phi = 1 - \frac{\omega^2h^2}{2}.$$

From that, you get that $x_n^{(h)} = x_{\mathrm{i}}\cos(n\phi)$. The method is thus stable.
Instead, if $\omega h > 2$, then the roots r_1 and r_2 are both real but their product still equals 1. Therefore, one of the roots is of absolute value larger than 1 and $x_n^{(h)}$ diverges. Therefore, the calculation becomes unstable if $h > 2/\omega$.

We end this section by summarizing the advantages and drawbacks of the velocity Verlet algorithm.

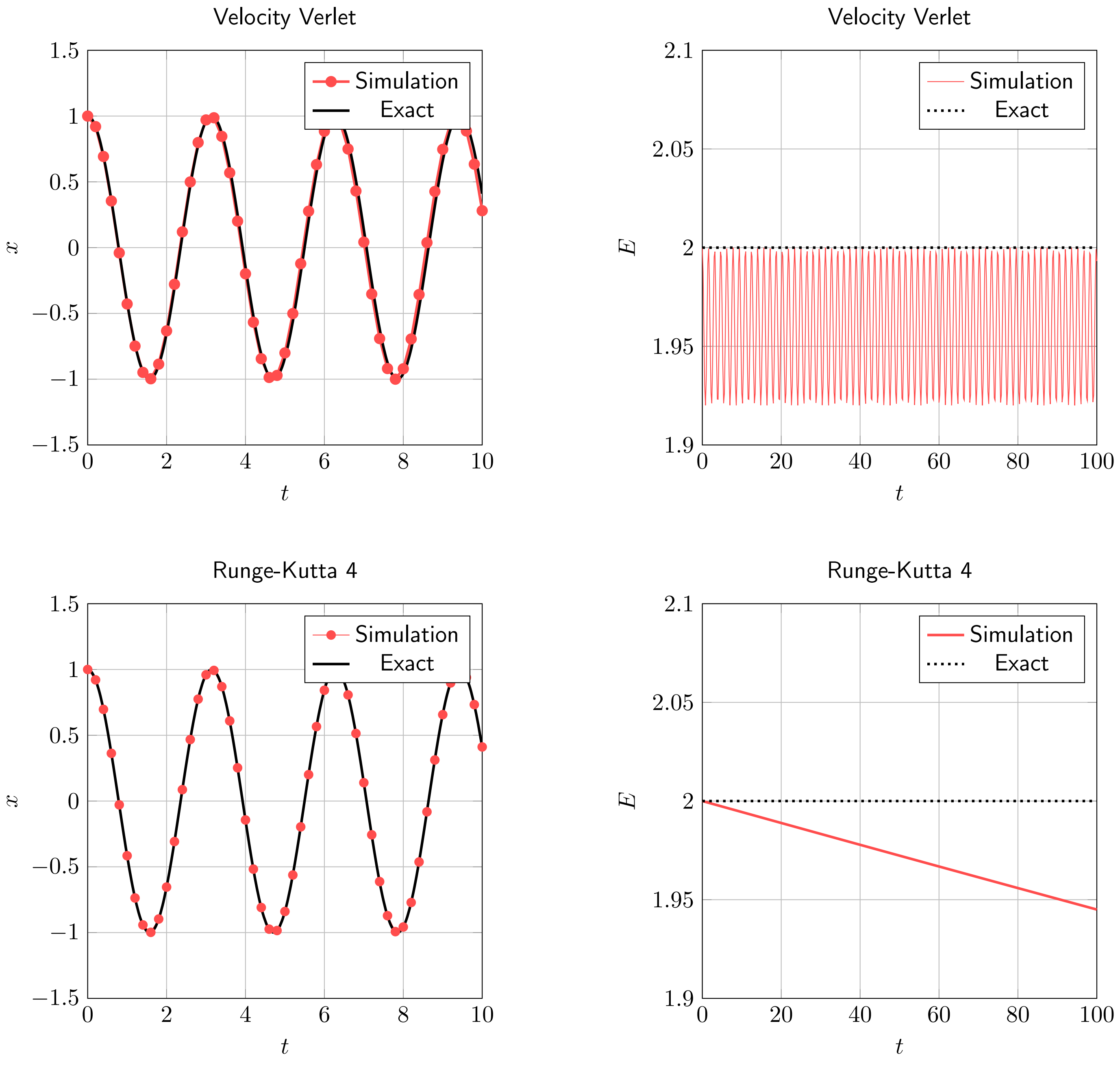

Figure 4: **Energy conservation with the velocity Verlet algorithm and the Runge-Kutta 4 method.** On the left panels, we show the result of the numerical integration of the ODE $\mathrm{d}^2x/\mathrm{d}t^2 = -\omega^2 x$ (harmonic oscillator) obtained with the velocity Verlet algorithm (top) or the Runge-Kutta 4 method (bottom). On the right panels, we show the energy $E = (1/2)m(\mathrm{d}x/\mathrm{d}t)^2 + (1/2)m\omega^2 x^2$ as a function of time for both methods on longer time intervals. While the Runge-Kutta 4 method has a better accuracy than the velocity Verlet method (the simulation points are close to the exact analytic result), energy systematically decreases although it should be conserved. Instead, energy oscillates close to the exact value with the velocity Verlet algorithm.

Advantages and Drawbacks of the velocity Verlet method
Advantages:

- The velocity Verlet method is **relatively simple** to implement and only requires to store the values of position and velocity at the previous step.
- The velocity Verlet method is **relatively fast** (few operations at each time step).

- The velocity Verlet method is **more precise** than the Euler method but **less precise** than the Runge-Kutta 4 method.
- The velocity Verlet method is **symplectic** unlike the Runge-Kutta 4 method: it approximately preserves conservation laws.

Drawbacks:

- The velocity Verlet method can be **unstable** when you have to solve stiff problems.

VIII. Boundary Value Problems

We end these lecture notes about the numerical integration of ODEs by dealing with the case of Boundary Value Problems (BVPs), which correspond to ODEs for which all the constants of integration are not given at the same time. As an illustration, we consider the following problem:

$$\begin{cases} \dfrac{\mathrm{d}^2\vec{x}}{\mathrm{d}t^2} = f\left(\vec{x},\, \dfrac{\mathrm{d}\vec{x}}{\mathrm{d}t}\right), \\ \vec{x}(t_\mathrm{i}) = \vec{x}_\mathrm{i}, \\ \vec{x}(t_\mathrm{f}) = \vec{x}_\mathrm{f}, \end{cases} \tag{20}$$

to be solved in the time interval $[t_\mathrm{i},\, t_\mathrm{f}]$. The problem is that we do not know the initial velocity. Therefore if we choose it at random, there is no guarantee that at the end of the integration we have $\vec{x}(t_\mathrm{f}) = \vec{x}_\mathrm{f}$. We propose a method to determine what initial condition on the velocity we should consider in order to meet the final condition $\vec{x}(t_\mathrm{f}) = \vec{x}_\mathrm{f}$. This method is called the **shooting method**.

Definition of the shooting method
To solve a BVP given by Eq. (20), we transform it into an optimization problem over IVPs. Said differently, we apply the following procedure.

1. Integrate the ODE up to time t_f with a initial guess for the initial velocity $\vec{v}_\mathrm{i} = \mathrm{d}\vec{x}/\mathrm{d}t(t_\mathrm{i})$.
2. Compare $\vec{x}(t_\mathrm{f})$ with $\vec{x}_\mathrm{f}$.
3. Adjust the initial velocity $\vec{v}_\mathrm{i}$ and iterate until you meet the condition $\vec{x}(t_\mathrm{f}) = \vec{x}_\mathrm{f}$ up to a given precision.

For the iteration, you can use a root-finding algorithm applied to the function $F : \vec{v}_\mathrm{i} \mapsto \vec{x}(t_\mathrm{f}) - \vec{x}_\mathrm{f}$, which associates to an initial velocity $\vec{v}_\mathrm{i}$ the value of the solution at time t_f minus the target value. Be careful that the velocity which allows you to meet the condition $\vec{x}(t_\mathrm{f}) = \vec{x}_\mathrm{f}$ is not necessarily unique (F may have multiple roots).

In the following, we propose an implementation for a one-dimensional problem with the bisection method as a root-finding algorithm (see Fig. 5). We assume that we know two values v_0 and v_0' of the initial velocity such that $F(v_0) > 0$ and $F(v_0') < 0$, where $F : v \mapsto x(t_\mathrm{f}) - x_\mathrm{f}$ with $x(t)$ the numerical solution of the Cauchy problem

$$\frac{\mathrm{d}^2x}{\mathrm{d}t^2} = f\left(x, \frac{\mathrm{d}x}{\mathrm{d}t}\right), \quad x(t_\mathrm{i}) = x_\mathrm{i}, \quad \frac{\mathrm{d}x}{\mathrm{d}t}(t_\mathrm{i}) = v.$$

Algorithm for the shooting method with a bisection root-finding scheme

```
$v_{\min} \leftarrow v_0$    ▷ Initiate the left bounding value $v_0$ of the velocity for which $F(v_{\min}) > 0$ to $v_0$.
$v_{\max} \leftarrow v_0'$    ▷ Initiate the right bounding value $v_{\max}$ of the velocity for which $F(v_{\max}) < 0$ to $v_0'$.
$v \leftarrow (v_{\min} + v_{\max})/2$    ▷ Choose the trial velocity as the middle between $v_{\min}$ and $v_{\max}$.
$x_{\mathrm{end}} \leftarrow \mathrm{FINAL\_VALUE\_IVP}(x_\mathrm{i}, v)$    ▷ Compute the value $x(t_\mathrm{f})$ of the solution at time $t_\mathrm{f}$ for initial conditions $x(t_\mathrm{i}) = x_\mathrm{i}$ and $\mathrm{d}x/\mathrm{d}t(t_\mathrm{i}) = v$.
```

```
while $|x_\text{end}/x_\text{f} - 1| > \epsilon$ do ▷ Iterate as long as the relative difference between $x(t_\text{f})$ and $x_\text{f}$ is larger than a given threshold $\epsilon$.
    if $x_\text{end} > x_\text{f}$ then
        $v_\text{min} \leftarrow v$                                   ▷ $F(v) > 0$ so update $v_\text{min}$.
    else
        $v_\text{max} \leftarrow v$                                   ▷ $F(v) < 0$ so update $v_\text{max}$.
    $v \leftarrow (v_\text{min} + v_\text{max})/2$ ▷ Choose the trial velocity as the middle between the updated values of $v_\text{min}$ and $v_\text{max}$.
    $x_\text{end} \leftarrow$ FINAL_VALUE_IVP$(x_\text{i}, v)$        ▷ Compute the new value of $x(t_\text{f})$.
```

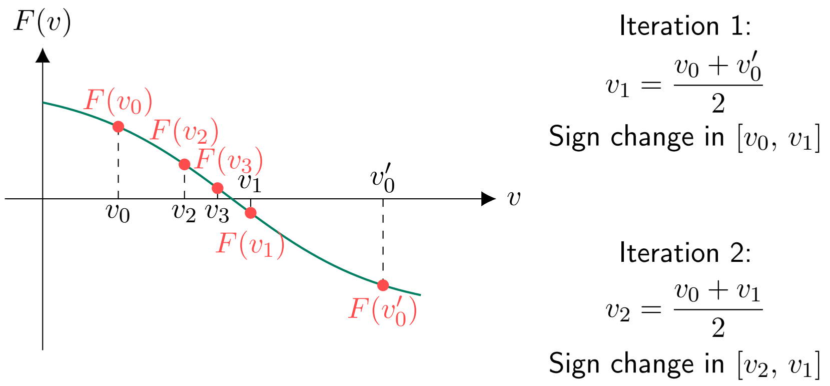
$F(v)$
v
$F(v_0)$
$F(v_2)$
$F(v_3)$
v_1
v_0'
v_0
v_2 v_3
$F(v_1)$
$F(v_0')$
Iteration 1:
$v_1 = \frac{v_0 + v_0'}{2}$
Sign change in $[v_0,\, v_1]$
Iteration 2:
$v_2 = \frac{v_0 + v_1}{2}$
Sign change in $[v_2,\, v_1]$

Figure 5: **Illustration of the bisection method to find the root of a function.** We look for the root of function F knowing that it lies in the range $[v_0,\, v_0']$ [with $F(v_0) > 0$ and $F(v_0') < 0$]. We start with $v_\text{min} = v_0$ and $v_\text{max} = v_0'$. At each step we choose v in the middle of the interval $[v_\text{min},\, v_\text{max}]$ and update v_min to v if $F(v) > 0$ or v_max to v if $F(v) < 0$. After few iterations, v converges to the root of F.