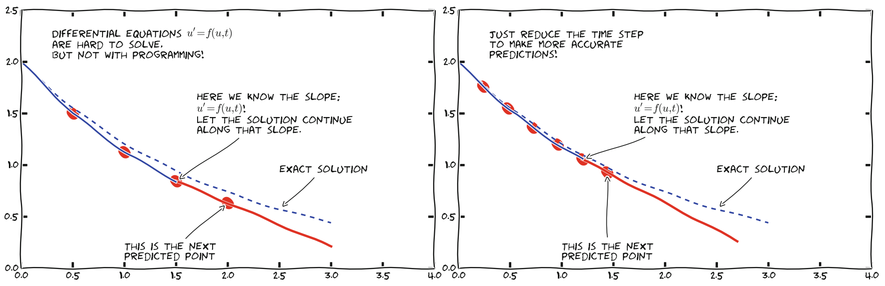
Solving Ordinary Differential Equations 8



Differential equations constitute one of the most powerful mathematical tools to understand and predict the behavior of dynamical systems in nature, engineering, and society. A dynamical system is some system with some state, usually expressed by a set of variables, that evolves in time. For example, an oscillating pendulum, the spreading of a disease, and the weather are examples of dynamical systems. We can use basic laws of physics, or plain intuition, to express mathematical rules that govern the evolution of a system in time. These rules take the form of *differential equations*.

You are probably well experienced with equations, at least equations like $ax + b = 0$ and $ax^2 + bx + c = 0$, where a, b and c are constants. Such equations are known as *algebraic equations*, and the unknowns are *numbers*. In a differential equation, the unknown is a *function*, and a differential equation will usually involve this function and one or more of its derivatives. When this function depends on a single independent variable, the equation is called an *ordinary differential equation* (ODE, plural: ODEs), which is different from a *partial differential equation* (PDE, plural: PDEs), in which the function depends on several independent variables (Chap. 9). As an example, $f'(x) = f(x)$ is a simple ODE (asking if there is any function f such that it equals its derivative—you might remember that e^x

© The Author(s) 2020

S. Linge, H. P. Langtangen, *Programming for Computations - Python*,
Texts in Computational Science and Engineering 15,
https://doi.org/10.1007/978-3-030-16877-3_8

is a candidate).[1] This is also an example of a *first-order* ODE, since the highest derivative appearing in the equation is a first derivative. When the highest derivative in an ODE is a second derivative, it is a *second-order* ODE, and so on (a similar terminology is used also for PDEs).

Order of ODE versus order of numerical solution method

Note that an ODE will have an *order* as just explained. This order, however, should not be confused with the *order* of a numerical solution method applied to solve that ODE. The latter refers to the convergence rate of the numerical solution method, addressed at the end of this chapter. We will present several such solution methods in this chapter, being first-order, second-order or fourth-order, for example (and a first-order ODE might, in principle, be solved by any of these methods).

The present chapter[2] starts out preparing for ODEs and the Forward Euler method, which is a first-order method. Then we explain in detail how to solve ODEs numerically with the Forward Euler method, both *single (scalar)* first-order ODEs and *systems* of first-order ODEs. After the "warm-up" application—filling of a water tank—aimed at the less mathematically trained reader, we demonstrate all the mathematical and programming details through two specific applications: population growth and spreading of diseases. The first few programs we write, are deliberately made very simple and similar, while we focus the computational ideas.

Then we turn to oscillating mechanical systems, which arise in a wide range of engineering situations. The differential equation is now of second order, and the Forward Euler method does not perform too well. This observation motivates the need for other solution methods, and we derive the Euler-Cromer scheme, the second- and fourth-order Runge-Kutta schemes, as well as a finite difference scheme (the latter to handle the second-order differential equation directly without reformulating it as a first-order system). The presentation starts with undamped free oscillations and then treats general oscillatory systems with possibly nonlinear damping, nonlinear spring forces, and arbitrary external excitation. Besides developing programs from scratch, we also demonstrate how to access ready-made implementations of more advanced differential equation solvers in Python.

As we progress with more advanced methods, we develop more sophisticated and reusable programs. In particular, we incorporate good testing strategies, which allows us to bring solid evidence of correct computations. Consequently, the beginning—with water tank, population growth and disease modeling examples—has a very gentle learning curve, while that curve gets significantly steeper towards the end of our section on oscillatory systems.

[1] Note that the notation for the derivative may differ. For example, $f'(x)$ could equally well be written as just f' (where the dependence on x is to be understood), or as $\frac{df}{dx}$.

[2] The reader should be aware of another excellent easy-to-read text by the late Prof. Langtangen, "Finite Difference Computing with Exponential Decay Models" (Springer, open access, 2016, https://www.springer.com/gp/book/9783319294384). It fits right in with the material on ODEs and PDEs of the present book.

8.1 Filling a Water Tank: Two Cases

If “ordinary differential equation” is not among your favorite expressions, then this section is for you.

Consider a 25 L tank that will be filled with water in two different ways. In the first case, *the water volume that enters the tank per time* (rate of volume increase) is piecewise constant, while in the second case, it is continuously increasing.

For each of these two cases, we are asked to develop a code that can predict (i.e., compute) how the total water volume V in the tank will develop with time t over a period of 3 s. Our calculations must be based on the information given: the initial volume of water (1L in both cases), and the volume of water entering the tank per time.

8.1.1 Case 1: Piecewise Constant Rate

In this simpler case, there is initially 1 L of water in the tank, i.e.,

$$V(0) = 1\,\mathrm{L},$$

while the rates of volume increase are given as:

$$\begin{aligned}
r &= 1\,\mathrm{L\,s^{-1}}, \qquad & 0\,\mathrm{s} < t < 1\,\mathrm{s}\,,\\
r &= 3\,\mathrm{L\,s^{-1}}, \qquad & 1\,\mathrm{s} \le t < 2\,\mathrm{s}\,,\\
r &= 7\,\mathrm{L\,s^{-1}}, \qquad & 2\,\mathrm{s} \le t \le 3\,\mathrm{s}\,.
\end{aligned}$$

Before turning to the programming, we should work out the exact solution by hand for this problem, since that is rather straight forward. Such a solution will of course be useful for verifying our implementation. In fact, comparing program output to these hand calculations should suffice for this particular problem.

Exact Solution by Hand Our reasoning goes like this: For each of the given subintervals (on the time axis), the total volume V of water in the tank will increase linearly. Thus, if we compute V after 1, 2 and 3 s, we will have what we need. We get

$$\begin{aligned}
V(0) &= 1\,\mathrm{L}\,,\\
V(1) &= 1\,\mathrm{L} + (1\,\mathrm{s})(1\,\mathrm{L\,s^{-1}}) = 2\,\mathrm{L}\,,\\
V(2) &= 2\,\mathrm{L} + (1\,\mathrm{s})(3\,\mathrm{L\,s^{-1}}) = 5\,\mathrm{L}\,,\\
V(3) &= 5\,\mathrm{L} + (1\,\mathrm{s})(7\,\mathrm{L\,s^{-1}}) = 12\,\mathrm{L}\,.
\end{aligned}$$

We also have what is required for plotting the exact solution, since we can just tell Python to plot the computed V values against t for $t = 0, 1, 2, 3$, and let Python fill in the straight lines in between these points. Therefore, the exact solution is ready and we may proceed to bring our reasoning into code.

Implementation and Performance A simple implementation may read (`rate_piecewise_constant.py`):

```
import numpy as np
import matplotlib.pyplot as plt

a = 0.0;  b = 3.0                     # time interval
N = 3                                 # number of time steps
dt = (b - a)/N                        # time step (s)
V_exact = [1.0, 2.0, 5.0, 12.0]       # exact volumes (L)
V = np.zeros(4)                       # numerically computed volume (L)
V[0] = 1                              # initial volume
r = np.zeros(3)                       # rates of volume increase (L/s)
r[0] = 1; r[1] = 3; r[2] = 7

for i in [0, 1, 2]:
    V[i+1] = V[i] + dt*r[i]

time = [0, 1, 2, 3]
plt.plot(time, V, 'bo-', time, V_exact, 'r')
plt.title('Case 1')
plt.legend(['numerical','exact'], loc='upper left')
plt.xlabel('t (s)')
plt.ylabel('V (L)')
plt.show()
```

As you can see, we have included the exact (hand computed) solution in the code, so that it gets plotted together with the numerical solution found by the program. The time step `dt` will become 1 s here, and running the code, produces the plot seen in Fig. 8.1. We note that the numerical and exact solution can not be distinguished in the plot.

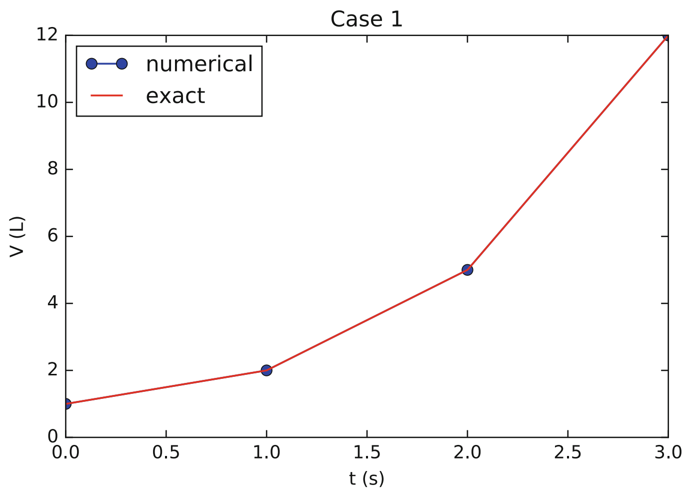


Fig. 8.1 Water volume in a tank as it develops with piecewise constant rate of volume increase

8.1.2 Case 2: Continuously Increasing Rate

This case is more tricky. As in the previous case, there is 1 L of water in the tank from the start. When the tank fills up, however, the rate of volume increase, r, is *always* equal to the current volume of water, i.e., $r = V$ (in units of $\mathrm{L\,s^{-1}}$). So, for example, at the time when there is 2 L in the tank, water enters the tank at $2\,\mathrm{L\,s^{-1}}$, when there is 2.1 L in the tank, water comes in at $2.1\,\mathrm{L\,s^{-1}}$, and so on. Thus, contrary to what we had in Case 1, r is not constant for any period of time within the 3 s interval, it increases continuously and gives us a steeper and steeper curve for $V(t)$. Writing this up as for Case 1, the information we have is

$$V(0) = 1\,\mathrm{L},$$

and, for the rate (in $\mathrm{L\,s^{-1}}$),

$$r(t) = V(t), \qquad 0\,\mathrm{s} < t \le 3\,\mathrm{s}\,.$$

Let us, for simplicity, assume that we also are given the exact solution in this case, which is $V(t) = e^t$. This allows us to easily check out the performance of any computational idea that we might try.

So, how can we compute the development of V, making it compare favorably to the given solution?

An Idea Clearly, we will be happy with an *approximately* correct solution, as long as the error can be made "small enough". In Case 1, we effectively computed connected straight line segments that matched the true development of V because of piecewise constant r values. Would it be possible to do something similar here in Case 2, i.e., compute straight line segments and use them as an *approximation* to the true solution curve? If so, it seems we could benefit from a very simple computational scheme! Let us pursue this idea further to see what comes out of it.

The First Time Step Considering the very first time step, we should realize that, since we are given the initial volume $V(0) = 1\,\mathrm{L}$, we do know the correct volume *and* correct rate at $t = 0$, since $r(t) = V(t)$. Thus, using this information and *pretending* that r stays constant as time increases, we will be able to compute a straight line segment for the very first time step (some Δt must be chosen). This straight line segment will then become tangent to the true solution curve when $t = 0$. The computed volume at the end of the first time step will have an error, but if our time step is not too large, the straight line segment will stay close to the true solution curve and the error in V should be "small".

The Second Time Step What about the second time step? Well, the volume we computed (with an error) at the end of the first time step, must now serve as the starting volume *and* ("constant") rate for the second time step. This allows us to compute an *approximation* to the volume also at the end of the second time step. If errors are "small", also the second straight line segment should be close to the true solution curve.

What About the Errors? We realize that, in this way, we can work our way all along the total time interval. Immediately, we suspect that the error may grow with the number of time steps, but since the total time interval is not too large, and since we may choose a *very* small time step on modern computers, this could still work!

Implementation and Performance Let us write down the code, which by choice gets very similar to the code in Case 1, and see how it performs. We realize that, for our strategy to work, the time steps should not be too large. However, during these initial investigations of ours, our aim is first and foremost to check out the computational idea. So, we pick a time step $\Delta t = 0.1$ s for a first try. A simple version of the code (`rate_exponential.py`) may then read:

```python
import numpy as np
import matplotlib.pyplot as plt

a = 0.0;  b = 3.0                    # time interval
N = 30                               # number of time steps
dt = (b - a)/N                       # time step (s)
V = np.zeros(N+1)                    # numerically computed volume (L)
V[0] = 1                             # initial volume

for i in range(0, N, 1):
    V[i+1] = V[i] + dt*V[i]          # ...r is V now

time_exact = np.linspace(a, b, 1000)
V_exact = np.exp(time_exact)         # make exact solution (for plotting)
time = np.linspace(0, 3, N+1)
plt.plot(time, V, 'bo-', time_exact, V_exact, 'r')
plt.title('Case 2')
plt.legend(['numerical','exact'], loc='upper left')
plt.xlabel('t (s)')
plt.ylabel('V (L)')
plt.show()
```

To plot the exact solution, we just picked 1000 points in time, which we consider "large enough" to get a representative curve. Compared to the code for Case 1, some more flexibility is introduced here, using `range` and `N` in the `for` loop header. Running the code gives the plot shown in Fig. 8.2.

This looks promising! Not surprisingly, the error grows with time, reaching about 2.64 L at the end. However, the time step is not particularly small, so we should expect much more accurate computations if Δt is reduced. We skip showing the plots,[3] but if we increase `N` from 30 to 300, the maximum error drops to 0.30 L, while an `N` value of $3 \cdot 10^6$ gives an error of $3 \cdot 10^{-5}$ L. It seems we are on to something!

[3] With smaller time steps, it becomes inappropriate to use filled circles on the graph for the numerical values. Thus, in the plot command, one should change `bo-` to, e.g., only `b`.

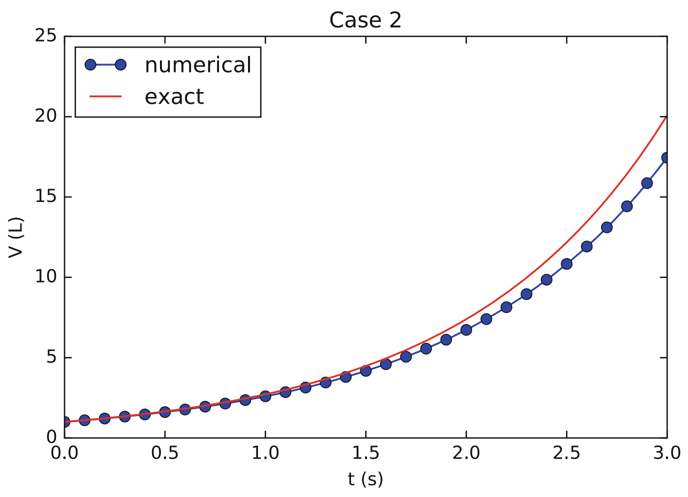


Fig. 8.2 Water volume in a tank as it develops with constantly increasing rate of volume change ($r = V$). The numerical computations use piecewise constant rates as an approximation to the true rate

The similar code structures used for Case 1 and Case 2 suggests that a more general (and improved) code may be written, applicable to both cases. As we move on with our next example, population growth, we will see how this can be done.

8.1.3 Reformulating the Problems as ODEs

Typically, the problems we solved in Case 1 and Case 2, would rather have been presented in "more proper mathematical language" as ODEs.

Case 1 When the rates were piecewise constant, we could have been requested to solve

$$\begin{aligned} V'(t) &= 1\,\mathrm{L\,s^{-1}}, && 0\,\mathrm{s} < t < 1\,\mathrm{s}\,, \\ V'(t) &= 3\,\mathrm{L\,s^{-1}}, && 1\,\mathrm{s} \le t < 2\,\mathrm{s}\,, \\ V'(t) &= 7\,\mathrm{L\,s^{-1}}, && 2\,\mathrm{s} \le t \le 3\,\mathrm{s}\,, \end{aligned}$$

with

$$V(0) = 1\,\mathrm{L},$$

where $V(0)$ is known as an *initial condition*.

Case 2 With a continuously increasing rate, we could have been asked to solve

$$V'(t) = V(t), \qquad V(0) = 1\,\mathrm{L}, \qquad 0\,\mathrm{s} < t \leq 3\,\mathrm{s}\,.$$

This particular ODE is very similar to the ODE we will address when we next turn to population growth (in fact, it may be seen as a special case of the latter).

The Forward Euler Method: A Brief Encounter If we had proceeded to solve these ODEs by something called the *Forward Euler method* (or *Euler's method*), we could (if we wanted) have written the solution codes exactly as they were developed above! Thus, we have already used the essentials of Euler's method without stating it.

In the following sections, the Forward Euler method will be thoroughly explained and elaborated on, while we demonstrate how the approach may be used to solve different ODEs numerically.

8.2 Population Growth: A First Order ODE

Our first real taste of differential equations regards modeling the growth of some population, such as a cell culture, an animal population, or a human population. The ideas even extend trivially to growth of money in a bank.

Let $N(t)$ be the number of individuals in the population at time t. How can we predict the evolution of N with time? Below we shall derive a differential equation whose solution is $N(t)$. The equation we will derive reads

$$N'(t) = rN(t), \tag{8.1}$$

where r is a number. Note that although N obviously is an integer in real life, we model N as a real-valued function. We choose to do this, because the solutions of differential equations are (normally continuous) real-valued functions. An integer-valued $N(t)$ in the model would lead to a lot of mathematical difficulties. Also, talking about, e.g., 2.5 individuals is no problem in mathematics, even though we must be a bit careful when applying this in a practical setting!

You may know, or find out, that the solution $N(t) = Ce^{rt}$, where C is any number. To make this solution unique, we need to fix C, which is done by prescribing the value of N at some time, usually at $t = 0$. If $N(0)$ is given as N_0, we get $N(t) = N_0e^{rt}$.

In general, a *differential equation model* consists of a *differential equation*, such as (8.1) *and* an *initial condition*, such as $N(0) = N_0$. With a known initial condition, the differential equation can be solved for the unknown function and the solution is unique.

It is very rare that we can find the solution of a differential equation as easy as the ODE in this example allows. Normally, one has to apply certain mathematical methods. Still, these methods can only handle some of the simplest differential equations. However, with numerical methods and a bit of programming, we can easily deal with almost any differential equation! This is exactly the topic of the present chapter.

8.2.1 Derivation of the Model

It can be instructive to show how an equation like (8.1) arises. Consider some population of an animal species and let $N(t)$ be the number of individuals in a certain spatial region, e.g. an island. We are not concerned with the spatial distribution of the animals, just the number of them in some region where there is no exchange of individuals with other regions. During a time interval Δt, some animals will die and some will be born. The numbers of deaths and births are expected to be proportional to N. For example, if there are twice as many individuals, we expect them to get twice as many newborns. In a time interval Δt, the net growth of the population will then be

$$N(t + \Delta t) - N(t) = \bar{b}N(t) - \bar{d}N(t),$$

where $\bar{b}N(t)$ is the number of newborns and $\bar{d}N(t)$ is the number of deaths. If we double Δt, we expect the proportionality constants $\bar{b}$ and $\bar{d}$ to double too, so it makes sense to think of $\bar{b}$ and $\bar{d}$ as proportional to Δt and “factor out” Δt. That is, we introduce $b = \bar{b}/\Delta t$ and $d = \bar{d}/\Delta t$ to be proportionality constants for newborns and deaths independent of Δt. Also, we introduce $r = b - d$, which is the net rate of growth of the population per time unit. Our model then becomes

$$N(t + \Delta t) - N(t) = \Delta t\, rN(t)\,. \tag{8.2}$$

Equation (8.2) is actually a computational model. Given $N(t)$, we can advance the population size by

$$N(t + \Delta t) = N(t) + \Delta t\, rN(t)\,.$$

This is called a *difference equation*. If we know $N(t)$ for some t, e.g., $N(0) = N_0$, we can compute

$$\begin{aligned}
N(\Delta t) &= N_0 + \Delta t\, rN_0,\\
N(2\Delta t) &= N(\Delta t) + \Delta t\, rN(\Delta t),\\
N(3\Delta t) &= N(2\Delta t) + \Delta t\, rN(2\Delta t),\\
&\vdots\\
N((k+1)\Delta t) &= N(k\Delta t) + \Delta t\, rN(k\Delta t),
\end{aligned}$$

where k is some arbitrary integer. A computer program can easily compute $N((k+1)\Delta t)$ for us with the aid of a little loop.

The initial condition

Observe that the computational formula cannot be started unless we have an initial condition!

The solution of $N' = rN$ is $N = Ce^{rt}$ for any constant C, and the initial condition is needed to fix C so the solution becomes unique. However, from a mathematical point of view, knowing $N(t)$ at any point t is sufficient as initial condition. Numerically, we more literally need an initial condition: we need to know a starting value at the left end of the interval in order to get the computational formula going.

In fact, we do not really need a computer in this particular case, since we see a repetitive pattern when doing hand calculations. This leads us to a mathematical formula for $N((k+1)\Delta t)$:

$$\begin{aligned} N((k+1)\Delta t) &= N(k\Delta t) + \Delta t\, rN(k\Delta t) = N(k\Delta t)(1+\Delta t\, r) \\ &= N((k-1)\Delta t)(1+\Delta t\, r)^2 \\ &\vdots \\ &= N_0(1+\Delta t\, r)^{k+1}\,. \end{aligned}$$

Rather than using (8.2) as a computational model directly, there is a strong tradition for deriving a differential equation from this difference equation. The idea is to consider a very small time interval Δt and look at the instantaneous growth as this time interval is shrunk to an infinitesimally small size. In mathematical terms, it means that we let $\Delta t \to 0$. As (8.2) stands, letting $\Delta t \to 0$ will just produce an equation $0 = 0$, so we have to divide by Δt and then take the limit:

$$\lim_{\Delta t \to 0} \frac{N(t+\Delta t) - N(t)}{\Delta t} = rN(t)\,.$$

The term on the left-hand side is actually the definition of the derivative $N'(t)$, so we have

$$N'(t) = rN(t),$$

which is the corresponding differential equation.

There is nothing in our derivation that forces the parameter r to be constant—it can change with time due to, e.g., seasonal changes or more permanent environmental changes.

Detour: Exact mathematical solution

If you have taken a course on mathematical solution methods for differential equations, you may want to recap how an equation like $N' = rN$ or $N' = r(t)N$ is solved. The *method of separation of variables* is the most convenient solution strategy in this case:

$$
\begin{aligned}
N' &= rN \\
\frac{dN}{dt} &= rN \\
\frac{dN}{N} &= rdt \\
\int_{N_0}^{N} \frac{dN}{N} &= \int_0^t rdt \\
\ln N - \ln N_0 &= \int_0^t r(t)dt \\
N &= N_0 \exp\left(\int_0^t r(t)dt\right),
\end{aligned}
$$

which for constant r results in $N = N_0 e^{rt}$. Note that $\exp(t)$ is the same as e^t.

As will be described later, r must in more realistic models depend on N. The method of separation of variables then requires to integrate $\int_{N_0}^{N} N/r(N)dN$, which quickly becomes non-trivial for many choices of $r(N)$. The only generally applicable solution approach is therefore a numerical method.

8.2.2 Numerical Solution: The Forward Euler (FE) Method

There is a huge collection of numerical methods for problems like (8.1), and in general any equation of the form $u' = f(u, t)$, where $u(t)$ is the unknown function in the problem, and f is some known formula of u and optionally t. In our case with population growth, i.e., (8.1), $u'(t)$ corresponds to $N'(t)$, while $f(u, t)$ corresponds to $rN(t)$.

We will first present a simple *finite difference method* solving $u' = f(u, t)$. The idea is fourfold:

1. Introduce $N_t + 1$ points in time, $t_0, t_1, \ldots, t_{N_t}$, for the relevant time interval. We seek the unknown u at these points in time, and introduce u^n as the numerical approximation to $u(t_n)$, see Fig. 8.3.
2. Utilize that the differential equation is valid at the mesh points.
3. Approximate derivatives by finite differences, see Fig. 8.4.
4. Formulate a computational algorithm that can compute a new value u^n based on previously computed values u^i, $i < n$.

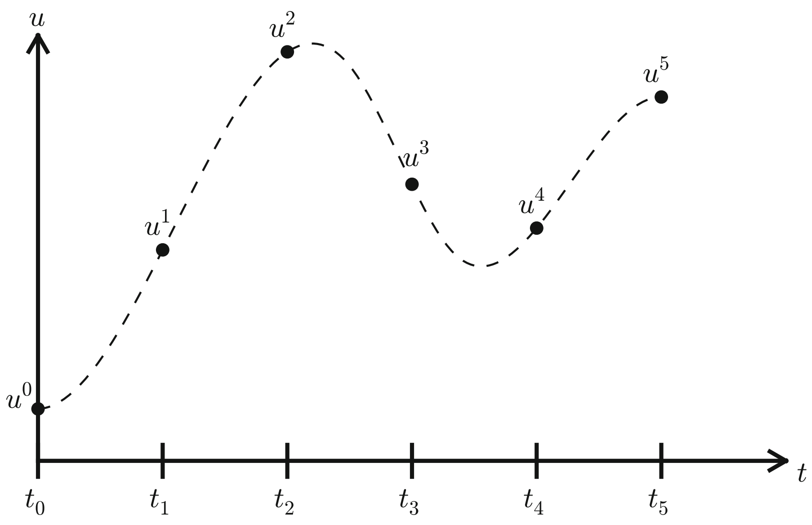


Fig. 8.3 Mesh in time with corresponding discrete values (unknowns)

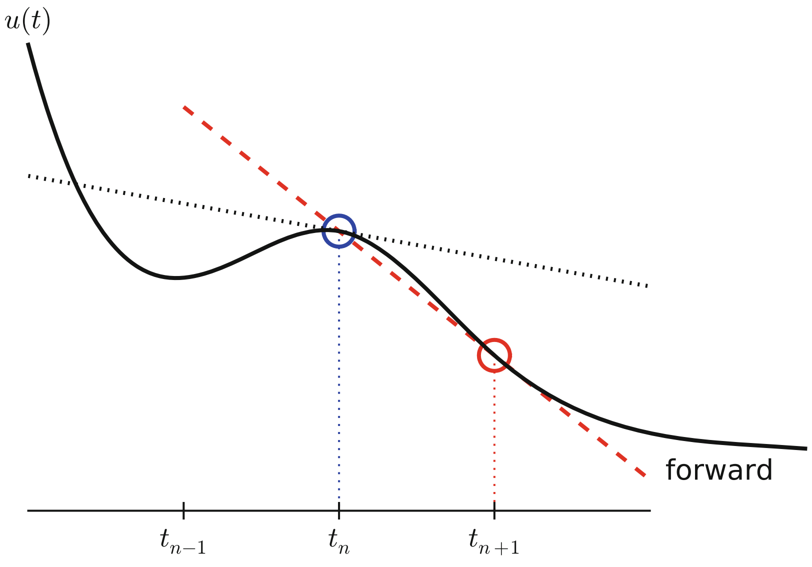


Fig. 8.4 Illustration of a forward difference approximation to the derivative

The collection of points $t_0, t_1, \ldots, t_{N_t}$ is called a *mesh* (or *grid*), while the corresponding u^n values[4] collectively are referred to as a *mesh function*.

Let us use an example to illustrate the given steps. First, we introduce the mesh, which often is *uniform*, meaning that the spacing between mesh points t_n and t_{n+1}

[4] To get an estimate of u in between the mesh points, one often assumes a straight line relationship between computed u^n values. Check out *linear interpolation*, e.g., on Wikipedia (https://en.wikipedia.org/wiki/Linear_interpolation).

is constant. This property implies that

$$t_n = n\Delta t, \quad n = 0, 1, \ldots, N_t\,.$$

Second, the differential equation is supposed to hold at the mesh points. Note that this is an approximation, because the differential equation is originally valid for all real values of t. We can express this property mathematically as

$$u'(t_n) = f(u^n, t_n), \quad n = 0, 1, \ldots, N_t\,.$$

For example, with our model equation $u' = ru$, we have the special case

$$u'(t_n) = ru^n, \quad n = 0, 1, \ldots, N_t,$$

or

$$u'(t_n) = r(t_n)u^n, \quad n = 0, 1, \ldots, N_t,$$

if r depends explicitly on t.

Third, derivatives are to be replaced by finite differences. To this end, we need to know specific formulas for how derivatives can be approximated by finite differences. One simple possibility is to use the definition of the derivative from any calculus book,

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

At an arbitrary mesh point t_n this definition can be written as

$$u'(t_n) = \lim_{\Delta t\to 0}\frac{u^{n+1}-u^n}{\Delta t}\,.$$

Instead of going to the limit $\Delta t \to 0$ we can use a small Δt, which yields a computable approximation to $u'(t_n)$:

$$u'(t_n) \approx \frac{u^{n+1}-u^n}{\Delta t}\,.$$

This is known as a *forward difference* since we go forward in time (u^{n+1}) to collect information in u to estimate the derivative. Figure 8.4 illustrates the idea. The error of the forward difference is proportional to Δt (often written as $\mathcal{O}(\Delta t)$, but we will not use this notation in the present book).

We can now plug in the forward difference in our differential equation sampled at the arbitrary mesh point t_n:

$$\frac{u^{n+1}-u^n}{\Delta t} = f(u^n, t_n), \tag{8.3}$$

or with $f(u, t) = ru$ in our special model problem for population growth,

$$\frac{u^{n+1} - u^n}{\Delta t} = ru^n \,. \tag{8.4}$$

If r depends on time, we insert $r(t_n) = r^n$ for r in this latter equation.

The fourth step is to derive a computational algorithm. Looking at (8.3), we realize that if u^n should be known, we can easily solve with respect to u^{n+1} to get a formula for u at the next time level t_{n+1}:

$$u^{n+1} = u^n + \Delta t f(u^n, t_n) \,. \tag{8.5}$$

Provided we have a known starting value, $u^0 = U_0$, we can use (8.5) to advance the solution by first computing u^1 from u^0, then u^2 from u^1, u^3 from u^2, and so forth.

Such an algorithm is called a *numerical scheme* for the differential equation. It is often written compactly as

$$u^{n+1} = u^n + \Delta t f(u^n, t_n), \quad u^0 = U_0, \quad n = 0, 1, \ldots, N_t - 1 \,. \tag{8.6}$$

This scheme is known as the *Forward Euler scheme*, also called *Euler's method*.

In our special population growth model, we have

$$u^{n+1} = u^n + \Delta t\, ru^n, \quad u^0 = U_0, \quad n = 0, 1, \ldots, N_t - 1 \,. \tag{8.7}$$

We may also write this model using the problem-specific symbol N instead of the generic u function:

$$N^{n+1} = N^n + \Delta t\, rN^n, \quad N^0 = N_0, \quad n = 0, 1, \ldots, N_t - 1 \,. \tag{8.8}$$

The observant reader will realize that (8.8) is nothing but the computational model (8.2) arising directly in the model derivation. The formula (8.8) arises, however, from a detour via a differential equation and a numerical method for the differential equation. This looks rather unnecessary! The reason why we bother to derive the differential equation model and then discretize it by a numerical method is simply that the discretization can be done in many ways, and we can create (much) more accurate and more computationally efficient methods than (8.8) or (8.6). This can be useful in many problems! Nevertheless, the Forward Euler scheme is intuitive and widely applicable, at least when Δt is chosen to be small.

The numerical solution between the mesh points

Our numerical method computes the unknown function u at discrete mesh points $t_1, t_2, \ldots, t_{N_t}$. What if we want to evaluate the numerical solution between the mesh points? The most natural choice is to *assume* a linear variation between the mesh points, see Fig. 8.5. This is compatible with the fact that when we plot the array $u^0, u^1, \ldots$ versus $t_0, t_1, \ldots$, a straight line is automatically drawn between the discrete points (unless we specify otherwise in the plot command).

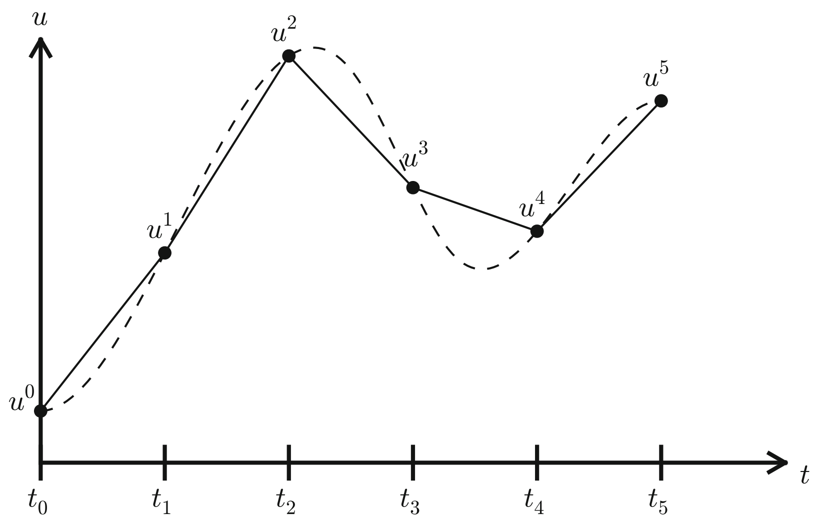


Fig. 8.5 The numerical solution at points can be extended by linear segments between the mesh points

8.2.3 Programming the FE Scheme; the Special Case

Let us compute (8.8) in a program. The input variables are N_0, Δt, r, and N_t. Note that we need to compute N_t new values $N^1, \ldots, N^{N_t}$. A total of $N_t + 1$ values are needed in an array representation of N^n, $n = 0, \ldots, N_t$.

Our first version of this program (growth1.py) is as simple as possible, and very similar to the codes we wrote previously for the water tank example:

```
import numpy as np
import matplotlib.pyplot as plt

N_0 = int(input('Give initial population size N_0: '))
r   = float(input('Give net growth rate r: '))
dt  = float(input('Give time step size: '))
N_t = int(input('Give number of steps: '))

t = np.linspace(0, N_t*dt, N_t+1)
N = np.zeros(N_t+1)

N[0] = N_0
for n in range(N_t):
    N[n+1] = N[n] + r*dt*N[n]

numerical_sol = 'bo' if N_t < 70 else 'b-'
plt.plot(t, N, numerical_sol, t, N_0*np.exp(r*t), 'r-')
plt.legend(['numerical', 'exact'], loc='upper left')
plt.xlabel('t'); plt.ylabel('N(t)')
filestem = 'growth1_{:d}steps'.format(N_t)
plt.savefig('{:s}.png'.format(filestem))
plt.savefig('{:s}.pdf'.format(filestem))
```

Note the compact assignment statement

```
numerical_sol = 'bo' if N_t < 70 else 'b-'
```

This is a one-line alternative to, e.g.,

```
if N_t < 70:
    numerical_sol = 'bo'
else:
    numerical_sol = 'b-'
```

Let us demonstrate a simulation where we start with 100 animals, a net growth rate of 10% (0.1) per time unit, which can be 1 month, and $t \in [0, 20]$ months. We may first try Δt of half a month (0.5), which implies $N_t = 40$. Figure 8.6 shows the results. The solid line is the exact solution, while the circles are the computed numerical solution. The discrepancy is clearly visible. What if we make Δt 10 times smaller? The result is displayed in Fig. 8.7, where we now use a solid line also for the numerical solution (otherwise, 400 circles would look very cluttered, so the program has a test on how to display the numerical solution, either as circles or a solid line). We can hardly distinguish the exact and the numerical solution. The computing time is also a fraction of a second on a laptop, so it appears that the Forward Euler method is sufficiently accurate for practical purposes. (This is not always true for large, complicated simulation models in engineering, so more sophisticated methods may be needed.)

It is also of interest to see what happens if we increase Δt to 2 months. The results in Fig. 8.8 indicate that this is an inaccurate computation.

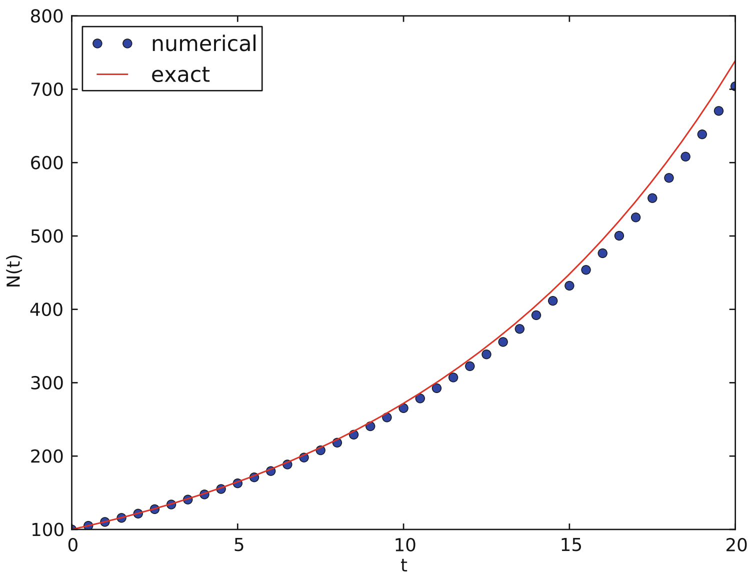


Fig. 8.6 Evolution of a population computed with time step 0.5 month

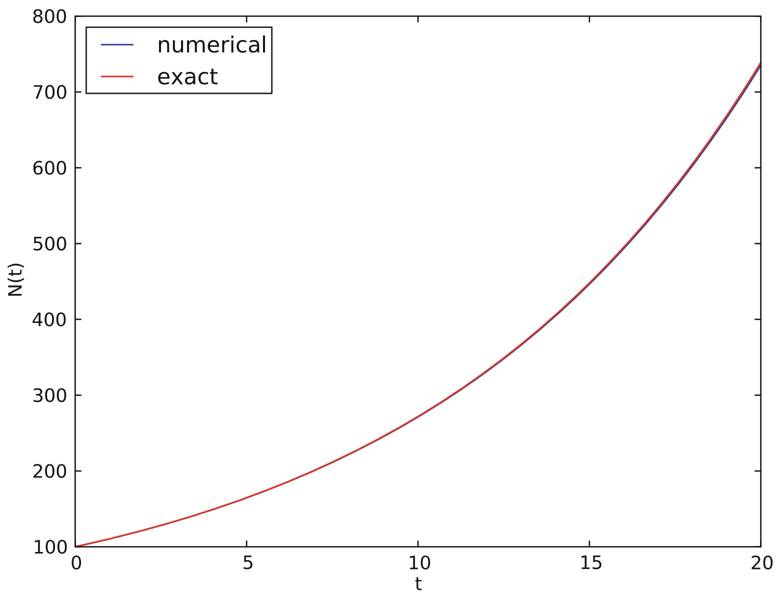


Fig. 8.7 Evolution of a population computed with time step 0.05 month

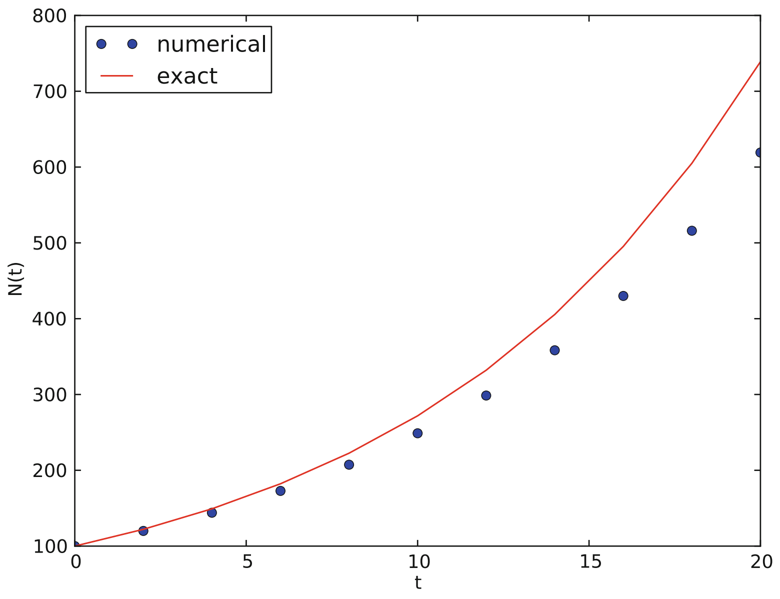


Fig. 8.8 Evolution of a population computed with time step 2 months

8.2.4 Understanding the Forward Euler Method

The good thing about the Forward Euler method is that it gives an understanding of what a differential equation is and a geometrical picture of how to construct the solution. The first idea is that we have already computed the solution up to some time point t_n. The second idea is that we want to progress the solution from t_n to t_{n+1} as a straight line.

We know that the line must go through the solution at t_n, i.e., the point (t_n, u^n). The differential equation tells us the slope of the line: $u'(t_n) = f(u^n, t_n) = ru^n$. That is, the differential equation gives a direct formula for the further *direction* of the solution curve. We can say that the differential equation expresses how the system (u) undergoes changes at a point.

There is a general formula for a straight line $y = ax + b$ with slope a that goes through the point (x_0, y_0): $y = a(x - x_0) + y_0$. Using this formula adapted to the present case, and evaluating the formula for t_{n+1}, results in

$$u^{n+1} = ru^n(t_{n+1} - t_n) + u^n = u^n + \Delta t\, ru^n,$$

which is nothing but the Forward Euler formula. You are now encouraged to do Exercise 8.2 to become more familiar with the geometric interpretation of the Forward Euler method.

8.2.5 Programming the FE Scheme; the General Case

Our previous program was just a simple main program without function definitions, tailored to a special differential equation. When programming mathematics, it is always good to consider a (large) class of problems and making a Python function to solve any problem that fits into the class. More specifically, we will make software for the class of differential equation problems of the form

$$u'(t) = f(u, t), \quad u = U_0,\ t \in [0, T],$$

for some given function f, and numbers U_0 and T. We also take the opportunity to illustrate what is commonly called a demo function. As the name implies, the purpose of such a function is solely to demonstrate how the function works (not to be confused with a test function, which does verification by use of `assert`). The Python function calculating the solution must take f, U_0, Δt, and T as input, find the corresponding N_t, compute the solution, and return an array with $u^0, u^1, \ldots, u^{N_t}$ and an array with $t_0, t_1, \ldots, t_{N_t}$. The Forward Euler scheme reads

$$u^{n+1} = u^n + \Delta t f(u^n, t_n), \quad n = 0, \ldots, N_t - 1\,.$$

The corresponding program may now take the form (file `ode_FE.py`):

```
import numpy as np
import matplotlib.pyplot as plt

def ode_FE(f, U_0, dt, T):
    N_t = int(round(T/dt))
    u = np.zeros(N_t+1)
    t = np.linspace(0, N_t*dt, len(u))
    u[0] = U_0
    for n in range(N_t):
        u[n+1] = u[n] + dt*f(u[n], t[n])
    return u, t

def demo_population_growth():
    """Test case: u'=r*u, u(0)=100."""
    def f(u, t):
        return 0.1*u

    u, t = ode_FE(f=f, U_0=100, dt=0.5, T=20)
    plt.plot(t, u, t, 100*np.exp(0.1*t))
    plt.show()

if __name__ == '__main__':
    demo_population_growth()
```

This program file, called `ode_FE.py`, is a reusable piece of code with a general `ode_FE` function that can solve any differential equation $u' = f(u, t)$ and a demo function for the special case $u' = 0.1u$, $u(0) = 100$. Observe that the call to the demo function is placed in a test block. This implies that the call is not active if `ode_FE` is imported as a module in another program, but active if `ode_FE.py` is run as a program.

The solution should be identical to what the `growth1.py` program produces with the same parameter settings ($r = 0.1$, $N_0 = 100$). This feature can easily be tested by inserting a print command, but a much better, automated verification is suggested in Exercise 8.2. You are strongly encouraged to take a "break" and do that exercise now.

Remark on the Use of `u` as Variable

In the `ode_FE` program, the variable `u` is used in different contexts. Inside the `ode_FE` function, `u` is an array, but in the `f(u,t)` function, as exemplified in the `demo_population_growth` function, the argument `u` is a number. Typically, we call `f` (in `ode_FE`) with the `u` argument as one element of the array `u` in the `ode_FE` function: `u[n]`.

8.2.6 A More Realistic Population Growth Model

Exponential growth of a population according the model $N' = rN$, with exponential solution $N = N_0e^{rt}$, is unrealistic in the long run because the resources needed to feed the population are finite. At some point there will not be enough resources and the growth will decline. A common model taking this effect into account assumes that r depends on the size of the population, N:

$$N(t + \Delta t) - N(t) = r(N(t))N(t)\,.$$

The corresponding differential equation becomes

$$N' = r(N)N\,.$$

The reader is strongly encouraged to repeat the steps in the derivation of the Forward Euler scheme and establish that we get

$$N^{n+1} = N^n + \Delta t\, r(N^n)N^n,$$

which computes as easy as for a constant r, since $r(N^n)$ is known when computing N^{n+1}. Alternatively, one can use the Forward Euler formula for the general problem $u' = f(u,t)$ and use $f(u,t) = r(u)u$ and replace u by N.

The simplest choice of $r(N)$ is a linear function, starting with some growth value $\bar{r}$ and declining until the population has reached its maximum, M, according to the available resources:

$$r(N) = \bar{r}(1 - N/M)\,.$$

In the beginning, $N \ll M$ and we will have exponential growth $e^{\bar{r}t}$, but as N increases, $r(N)$ decreases, and when N reaches M, $r(N) = 0$ so there is no more growth and the population remains at $N(t) = M$. This linear choice of $r(N)$ gives rise to a model that is called the *logistic model*. The parameter M is known as the *carrying capacity* of the population.

Let us run the logistic model with aid of the ode_FE function. We choose $N(0) = 100$, $\Delta t = 0.5$ month, $T = 60$ months, $r = 0.1$, and $M = 500$. The complete program, called logistic.py, is basically a call to ode_FE:

```
from ode_FE import ode_FE
import matplotlib.pyplot as plt

for dt, T in zip((0.5, 20), (60, 100)):
    u, t = ode_FE(f=lambda u, t: 0.1*(1 - u/500.)*u, \
                                U_0=100, dt=dt, T=T)
    plt.figure()  # Make separate figures for each pass in the loop
    plt.plot(t, u, 'b-')
    plt.xlabel('t'); plt.ylabel('N(t)')
    plt.savefig('tmp_{:g}.png'.format(dt))
    plt.savefig('tmp_{:g}.pdf'.format(dt))
```

Figure 8.9 shows the resulting curve. We see that the population stabilizes around $M = 500$ individuals. A corresponding exponential growth would reach $N_0 e^{rt} = 100e^{0.1\cdot 60} \approx 40,300$ individuals!

What happens if we use “large” Δt values here? We may set $\Delta t = 20$ and $T = 100$. Now the solution, seen in Fig. 8.10, oscillates and is hence qualitatively wrong, because one can prove that the exact solution of the differential equation is monotone.

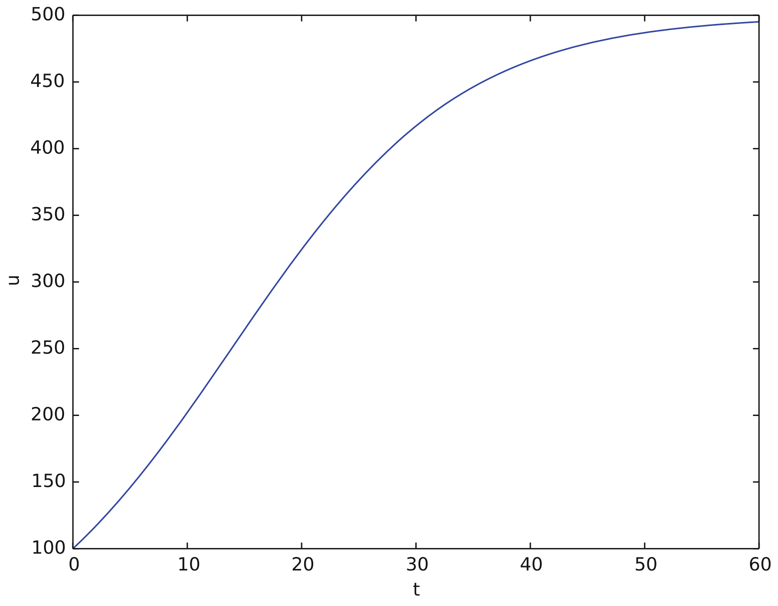


Fig. 8.9 Logistic growth of a population

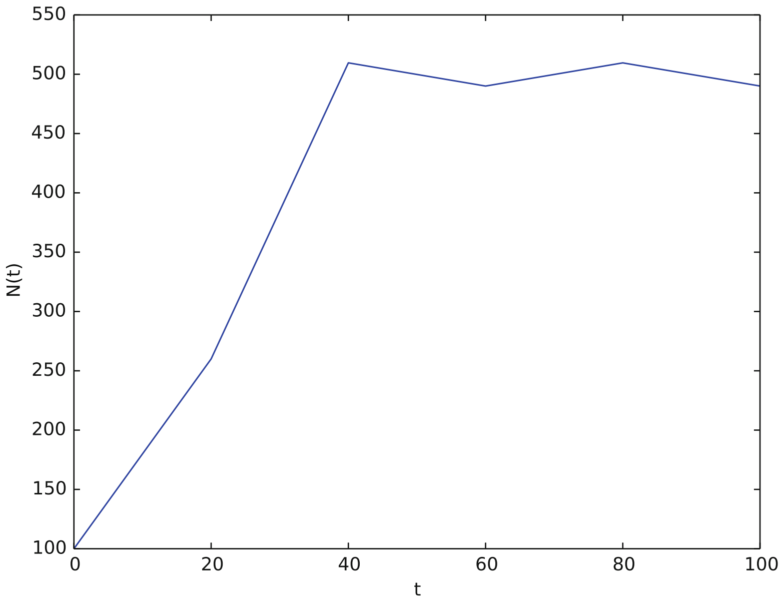


Fig. 8.10 Logistic growth with large time step

Remark on the world population

The number of people on the planet (http://en.wikipedia.org/wiki/Population_growth) follows the model $N' = r(t)N$, where the net reproduction $r(t)$ varies with time and has decreased since its top in 1990. The current world value of r is 1.2%, and it is difficult to predict future values. At the moment, the predictions of the world population point to a growth to 9.6 billion before declining.

This example shows the limitation of a differential equation model: we need to know all input parameters, including $r(t)$, in order to predict the future. It is seldom the case that we know all input parameters. Sometimes knowledge of the solution from measurements can help estimate missing input parameters.

8.2.7 Verification: Exact Linear Solution of the Discrete Equations

How can we verify that the programming of an ODE model is correct? One way, is to compute convergence rates with a solver and confirm that the rates are according to expectations. We address convergence rates for ODE solvers later (in Sect. 8.5) and will then show how a corresponding test function for the `ode_FE` solver may be written.

The best verification method, however, is to find a problem where there are no unknown numerical approximation errors, because we can then compare the exact solution of the problem with the result produced by our implementation and expect the difference to be within a very small tolerance. We shall base a unit test on this idea and implement a corresponding *test function* (see Sect. 6.6.4) for automatic verification of our implementation.

It appears that most numerical methods for ODEs will exactly reproduce a solution u that is linear in t. We may therefore set $u = at + b$ and choose any f whose derivative is a. The choice $f(u,t) = a$ is very simple, but we may add anything that is zero, e.g.,

$$f(u,t) = a + (u - (at + b))^m .$$

This is a valid $f(u,t)$ for any a, b, and m. The corresponding ODE looks highly non-trivial, however:

$$u' = a + (u - (at + b))^m .$$

Using the general `ode_FE` function in `ode_FE.py`, we may write a proper test function as follows (in file `test_ode_FE_exact_linear.py`):

```
def test_ode_FE():
    """Test that a linear u(t)=a*t+b is exactly reproduced."""
    def exact_solution(t):
        return a*t + b
    def f(u, t):  # ODE
        return a + (u - exact_solution(t))**m

    a = 4
    b = -1
    m = 6

    dt = 0.5
    T = 20.0

    u, t = ode_FE(f, exact_solution(0), dt, T)
    diff = abs(exact_solution(t) - u).max()
    tol = 1E-15           # Tolerance for float comparison
    success = diff < tol
    assert success
```

As a measure of the error, we have here simply used the maximum error (picked out by a call to `max` and assigned to `diff`).

Recall that test functions should start with the name `test_`, have no arguments, and formulate the test as a boolean expression `success` that is `True` if the test passes and `False` if it fails. Test functions should make the test as `assert success` (here `success` can also be a boolean expression as in `assert diff < tol`).

Observe that we cannot compare `diff` to zero, which is what we mathematically expect, because `diff` is a floating-point variable that most likely contains small rounding errors. Therefore, we must compare `diff` to zero with a tolerance, here 10^{-15}.

You are encouraged to do Exercise 8.3 where the goal is to make a test function for a verification based on comparison with hand-calculated results for a few time steps.

8.3 Spreading of Disease: A System of First Order ODEs

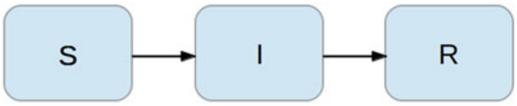
Our aim with this section is to show in detail how one can apply mathematics and programming to solve a system of first-order ODEs. We will do this as we investigate the spreading of disease. The mathematical model is now a system of three differential equations with three unknown functions. To derive such a model, we can use mainly intuition, so no specific background knowledge of diseases is required.

8.3.1 Spreading of Flu

Imagine a boarding school out in the country side. This school is a small and closed society. Suddenly, one or more of the pupils get the flu. We expect that the flu may spread quite effectively or die out. The question is how many of the pupils and the school's staff will be affected. Some quite simple mathematics can help us to achieve insight into the dynamics of how the disease spreads.

Let the mathematical function $S(t)$ count how many individuals, at time t, that have the possibility to get infected. Here, t may count hours or days, for instance. These individuals make up a category called susceptibles, labeled as S. Another category, I, consists of the individuals that are infected. Let $I(t)$ count how many there are in category I at time t. An individual having recovered from the disease is assumed to gain immunity. There is also a small possibility that an infected will die. In either case, the individual is moved from the I category to a category we call the removed category, labeled with R. We let $R(t)$ count the number of individuals in the R category at time t. Those who enter the R category, cannot leave this category.

To summarize, the spreading of this disease is essentially the dynamics of moving individuals from the S to the I and then to the R category:



We can use mathematics to more precisely describe the exchange between the categories. The fundamental idea is to describe the changes that take place during a small time interval, denoted by Δt.

Our disease model is often referred to as a *compartment model*, where quantities are shuffled between compartments (here a synonym for categories) according to some rules. The rules express *changes* in a small time interval Δt, and from these changes we can let Δt go to zero and obtain derivatives. The resulting equations then go from difference equations (with finite Δt) to differential equations ($\Delta t \to 0$).

We introduce a uniform mesh in time, $t_n = n\Delta t$, $n = 0, \ldots, N_t$, and seek S at the mesh points. The numerical approximation to S at time t_n is denoted by S^n. Similarly, we seek the unknown values of $I(t)$ and $R(t)$ at the mesh points and introduce a similar notation I^n and R^n for the approximations to the exact values $I(t_n)$ and $R(t_n)$.

In the time interval Δt we know that some people will be infected, so S will decrease. We shall soon argue by mathematics that there will be $\beta \Delta t SI$ new infected individuals in this time interval, where β is a parameter reflecting how easy people get infected during a time interval of unit length. If the loss in S is $\beta \Delta t SI$, we have that the change in S is

$$S^{n+1} - S^n = -\beta \Delta t S^n I^n \,. \tag{8.9}$$

Dividing by Δt and letting $\Delta t \to 0$, makes the left-hand side approach $S'(t_n)$ such that we obtain a differential equation

$$S' = -\beta SI \,. \tag{8.10}$$

The reasoning in going from the difference equation (8.9) to the differential equation (8.10) follows exactly the steps explained in Sect. 8.2.1.

Before proceeding with how I and R develops in time, let us explain the formula $\beta \Delta t SI$. We have S susceptibles and I infected people. These can make up SI pairs. Now, suppose that during a time interval T we measure that m actual pairwise meetings do occur among n theoretically possible pairings of people from the S and I categories. The probability that people meet in pairs during a time T is (by the empirical frequency definition of probability) equal to m/n, i.e., the number of successes divided by the number of possible outcomes. From such statistics we normally derive quantities expressed per unit time, i.e., here we want the probability per unit time, μ, which is found from dividing by T: $\mu = m/(nT)$.

Given the probability μ, the expected number of meetings per time interval of SI possible pairs of people is (from basic statistics) μSI. During a time interval Δt, there will be $\mu SI\Delta t$ expected number of meetings between susceptibles and infected people such that the virus may spread. Only a fraction of the $\mu\Delta tSI$ meetings are effective in the sense that the susceptible actually becomes infected. Counting that m people get infected in n such pairwise meetings (say 5 are infected from 1000 meetings), we can estimate the probability of being infected as $p = m/n$. The expected number of individuals in the S category that in a time interval Δt catch the virus and get infected is then $p\mu\Delta tSI$. Introducing a new constant $\beta = p\mu$ to save some writing, we arrive at the formula $\beta\Delta tSI$.

The value of β must be known in order to predict the future with the disease model. One possibility is to estimate p and μ from their meanings in the derivation above. Alternatively, we can observe an "experiment" where there are S_0 susceptibles and I_0 infected at some point in time. During a time interval T we count that N susceptibles have become infected. Using (8.9) as a rough approximation of how S has developed during time T (and now T is not necessarily small, but we use (8.9) anyway), we get

$$N = \beta TS_0I_0 \quad\Rightarrow\quad \beta = \frac{N}{TS_0I_0}\,. \tag{8.11}$$

We need an additional equation to describe the evolution of $I(t)$. Such an equation is easy to establish by noting that the loss in the S category is a corresponding gain in the I category. More precisely,

$$I^{n+1} - I^n = \beta\Delta tS^nI^n\,. \tag{8.12}$$

However, there is also a loss in the I category because people recover from the disease. Suppose that we can measure that m out of n individuals recover in a time period T (say 10 of 40 sick people recover during a day: $m = 10$, $n = 40$, $T = 24$ h). Now, $\gamma = m/(nT)$ is the probability that one individual recovers in a unit time interval. Then (on average) $\gamma\Delta tI$ infected will recover in a time interval Δt. This quantity represents a loss in the I category and a gain in the R category. We can therefore write the total change in the I category as

$$I^{n+1} - I^n = \beta\Delta tS^nI^n - \gamma\Delta tI^n\,. \tag{8.13}$$

The change in the R category is simple: there is always an increase got from the I category:

$$R^{n+1} - R^n = \gamma\Delta tI^n\,. \tag{8.14}$$

Since there is no loss in the R category (people are either recovered and immune, or dead), we are done with the modeling of this category. In fact, we do not strictly need Eq. (8.14) for R, but extensions of the model later will need an equation for R.

Dividing by Δt in (8.13) and (8.14) and letting $\Delta t \rightarrow 0$, results in the corresponding differential equations

$$I' = \beta S I - \gamma I, \tag{8.15}$$

and

$$R' = \gamma I\,. \tag{8.16}$$

To summarize, we have derived three difference equations and three differential equations, which we list here for easy reference. The difference equations are:

$$S^{n+1} = S^n - \beta \Delta t S^n I^n, \tag{8.17}$$

$$I^{n+1} = I^n + \beta \Delta t S^n I^n - \gamma \Delta t I^n, \tag{8.18}$$

$$R^{n+1} = R^n + \gamma \Delta t I^n\,. \tag{8.19}$$

Note that we have isolated the new unknown quantities S^{n+1}, I^{n+1}, and R^{n+1} on the left-hand side, such that these can readily be computed if S^n, I^n, and R^n are known. To get such a procedure started, we need to know S^0, I^0, R^0. Obviously, we also need to have values for the parameters β and γ.

The three differential equations are:

$$S' = -\beta S I, \tag{8.20}$$

$$I' = \beta S I - \gamma I, \tag{8.21}$$

$$R' = \gamma I\,. \tag{8.22}$$

This differential equation model (and also its discrete counterpart above) is known as an *SIR model*. The input data to the differential equation model consist of the parameter values for β and γ, as well as the initial conditions $S(0) = S_0$, $I(0) = I_0$, and $R(0) = R_0$.

8.3.2 A FE Method for the System of ODEs

Let us apply the same principles as we did in Sect. 8.2.2 to discretize the differential equation system by the Forward Euler method. We already have a time mesh and time-discrete quantities S^n, I^n, R^n, $n = 0, \ldots, N_t$. The three differential equations are assumed to be valid at the mesh points. At the point t_n we then have

$$S'(t_n) = -\beta S(t_n) I(t_n), \tag{8.23}$$

$$I'(t_n) = \beta S(t_n) I(t_n) - \gamma I(t_n), \tag{8.24}$$

$$R'(t_n) = \gamma I(t_n), \tag{8.25}$$

for $n = 0, 1, \ldots, N_t$. This is an approximation since the differential equations are originally valid at all times t (usually in some finite interval $[0, T]$). Using forward finite differences for the derivatives results in an additional approximation,

$$\frac{S^{n+1} - S^n}{\Delta t} = -\beta S^n I^n, \tag{8.26}$$

$$\frac{I^{n+1} - I^n}{\Delta t} = \beta S^n I^n - \gamma I^n, \tag{8.27}$$

$$\frac{R^{n+1} - R^n}{\Delta t} = \gamma I^n . \tag{8.28}$$

As we can see, these equations are identical to the difference equations that naturally arise in the derivation of the model. However, other numerical methods than the Forward Euler scheme will result in slightly different difference equations.

8.3.3 Programming the FE Scheme; the Special Case

The computation of (8.26)–(8.28) can be readily made in a computer program SIR1.py:

```
import numpy as np
import matplotlib.pyplot as plt

# Time unit: 1 h
beta = 10./(40*8*24)
gamma = 3./(15*24)
dt = 0.1             # 6 min
D = 30               # Simulate for D days
N_t = int(D*24/dt)   # Corresponding no of time steps

t = np.linspace(0, N_t*dt, N_t+1)
S = np.zeros(N_t+1)
I = np.zeros(N_t+1)
R = np.zeros(N_t+1)

# Initial condition
S[0] = 50
I[0] = 1
R[0] = 0

# Step equations forward in time
for n in range(N_t):
    S[n+1] = S[n] - dt*beta*S[n]*I[n]
    I[n+1] = I[n] + dt*beta*S[n]*I[n] - dt*gamma*I[n]
    R[n+1] = R[n] + dt*gamma*I[n]

fig = plt.figure()
l1, l2, l3 = plt.plot(t, S, t, I, t, R)
fig.legend((l1, l2, l3), ('S', 'I', 'R'), 'center right')
plt.xlabel('hours')
plt.savefig('tmp.pdf'); plt.savefig('tmp.png')
plt.show()
```

This program was written to investigate the spreading of flu at the mentioned boarding school, and the reasoning for the specific choices β and γ goes as follows. At some other school where the disease has already spread, it was observed that in the beginning of a day there were 40 susceptibles and 8 infected, while the numbers were 30 and 18, respectively, 24 h later. Using 1 h as time unit, we then have from (8.11) that $\beta = 10/(40 \cdot 8 \cdot 24)$. Among 15 infected, it was observed that 3 recovered during a day, giving $\gamma = 3/(15 \cdot 24)$. Applying these parameters to a new case where there is one infected initially and 50 susceptibles, gives the graphs in Fig. 8.11. These graphs are just straight lines between the values at times $t_i = i\Delta t$ as computed by the program. We observe that S reduces as I and R grows. After about 30 days everyone has become ill and recovered again.

We can experiment with β and γ to see whether we get an outbreak of the disease or not. Imagine that a "wash your hands" campaign was successful and that the other school in this case experienced a reduction of β by a factor of 5. With this lower β the disease spreads very slowly so we simulate for 60 days. The curves appear in Fig. 8.12.

8.3.4 Outbreak or Not

Looking at the equation for I, it is clear that we must have $\beta SI - \gamma I > 0$ for I to increase. When we start the simulation it means that

$$\beta S(0)I(0) - \gamma I(0) > 0,$$

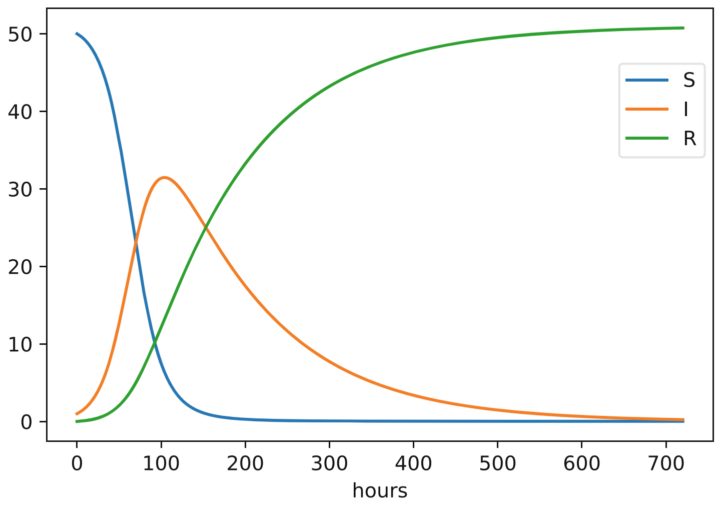


Fig. 8.11 Natural evolution of flu at a boarding school

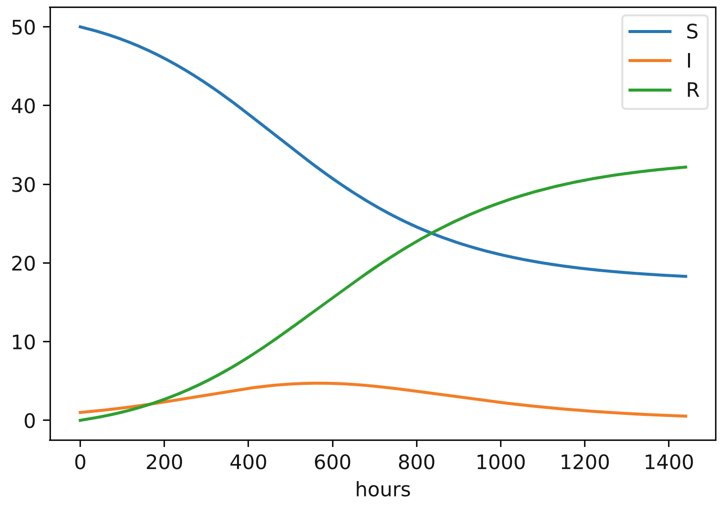


Fig. 8.12 Small outbreak of flu at a boarding school (β is much smaller than in Fig. 8.11)

or simpler

$$\frac{\beta S(0)}{\gamma} > 1 \tag{8.29}$$

to increase the number of infected people and accelerate the spreading of the disease. You can run the `SIR1.py` program with a smaller β such that (8.29) is violated and observe that there is no outbreak.

The power of mathematical modeling

The reader should notice our careful use of words in the previous paragraphs. We started out with modeling a very specific case, namely the spreading of flu among pupils and staff at a boarding school. With purpose we exchanged words like pupils and flu with more neutral and general words like *individuals* and *disease*, respectively. Phrased equivalently, we raised the abstraction level by moving from a specific case (flu at a boarding school) to a more general case (disease in a closed society). Very often, when developing mathematical models, we start with a specific example and see, through the modeling, that what is going on of essence in this example also will take place in many similar problem settings. We try to incorporate this generalization in the model so that the model has a much wider application area than what we aimed at in the beginning. This is the very power of mathematical modeling: by solving one specific case we have often developed more generic tools that can readily be applied to solve seemingly different problems. The next sections will give substance to this assertion.

8.3.5 Abstract Problem and Notation

When we had a specific differential equation with one unknown, we quickly turned to an abstract differential equation written in the generic form $u' = f(u, t)$. We refer to such a problem as a *scalar ODE*. A specific equation corresponds to a specific choice of the formula $f(u, t)$ involving u and (optionally) t.

It is advantageous to also write a system of differential equations in the same abstract notation,

$$u' = f(u, t),$$

but this time it is understood that u is a vector of functions and f is also vector. We say that $u' = f(u, t)$ is a *vector ODE* or *system of ODEs* in this case. For the SIR model we introduce the two 3-vectors, one for the unknowns,

$$u = (S(t), I(t), R(t)),$$

and one for the right-hand side functions,

$$f(u, t) = (-\beta SI, \beta SI - \gamma I, \gamma I)\,.$$

The equation $u' = f(u, t)$ means setting the two vectors equal, i.e., the components must be pairwise equal. Since $u' = (S', I', R')$, we get that $u' = f$ implies

$$\begin{aligned} S' &= -\beta SI, \\ I' &= \beta SI - \gamma I, \\ R' &= \gamma I\,. \end{aligned}$$

The generalized short notation $u' = f(u, t)$ is very handy since we can derive numerical methods and implement software for this abstract system and in a particular application just identify the formulas in the f vector, implement these, and call functionality that solves the differential equation system.

8.3.6 Programming the FE Scheme; the General Case

In Python code, the Forward Euler step

$$u^{n+1} = u^n + \Delta t f(u^n, t_n),$$

being a scalar or a vector equation, can be coded as

```
u[n+1] = u[n] + dt*f(u[n], t[n])
```

both in the scalar and vector case. In the vector case, `u[n]` is a one-dimensional `numpy` array of length $m + 1$ holding the mathematical quantity u^n, and the Python function `f` must return a `numpy` array of length $m + 1$. Then the expression `u[n] + dt*f(u[n], t[n])` is an array plus a scalar times an array.

For all this to work, the complete numerical solution must be represented by a two-dimensional array, created by `u = zeros((N_t+1, m+1))`. The first index counts the time points and the second the components of the solution vector at one time point. That is, `u[n,i]` corresponds to the mathematical quantity u_i^n. When we use only one index, as in `u[n]`, this is the same as `u[n,:]` and picks out all the components in the solution at the time point with index `n`. Then the assignment `u[n+1] = ...` becomes correct because it is actually an in-place assignment `u[n+1, :] = ...`. The nice feature of these facts is that the same piece of Python code works for both a scalar ODE and a system of ODEs!

The `ode_FE` function for the vector ODE is placed in the file `ode_system_FE.py` and was written as follows:

```python
import numpy as np
import matplotlib.pyplot as plt

def ode_FE(f, U_0, dt, T):
    N_t = int(round(T/dt))
    # Ensure that any list/tuple returned from f_ is wrapped as array
    f_ = lambda u, t: np.asarray(f(u, t))
    u = np.zeros((N_t+1, len(U_0)))
    t = np.linspace(0, N_t*dt, len(u))
    u[0] = U_0
    for n in range(N_t):
        u[n+1] = u[n] + dt*f_(u[n], t[n])
    return u, t
```

The line `f_ = lambda ...` needs an explanation. For a user, who just needs to define the f in the ODE system, it is convenient to insert the various mathematical expressions on the right-hand sides in a list and return that list. Obviously, we could demand the user to convert the list to a `numpy` array, but it is so easy to do a general such conversion in the `ode_FE` function as well. To make sure that the result from `f` is indeed an array that can be used for array computing in the formula `u[n] + dt*f(u[n], t[n])`, we introduce a new function `f_` that calls the user's `f` and sends the results through the `numpy` function `asarray`, which ensures that its argument is converted to a `numpy` array (if it is not already an array).

Note also the extra parenthesis required when calling `zeros` with two indices.

Let us show how the previous SIR model can be solved using the new general `ode_FE` that can solve *any* vector ODE. The user's `f(u, t)` function takes a vector `u`, with three components corresponding to S, I, and R as argument, along with the current time point `t[n]`, and must return the values of the formulas of the right-hand sides in the vector ODE. An appropriate implementation is

```python
def f(u, t):
    S, I, R = u
    return [-beta*S*I, beta*S*I - gamma*I, gamma*I]
```

Note that the `S`, `I`, and `R` values correspond to S^n, I^n, and R^n. These values are then just inserted in the various formulas in the vector ODE. Here we collect the values in a list since the `ode_FE` function will anyway wrap this list in an array. We could, of course, returned an array instead:

```python
def f(u, t):
    S, I, R = u
    return array([-beta*S*I, beta*S*I - gamma*I, gamma*I])
```

The list version looks a bit nicer, so that is why we prefer a list and rather introduce `f_ = lambda u, t: asarray(f(u,t))` in the general `ode_FE` function.

We can now show a function that runs the previous SIR example, while using the generic `ode_FE` function:

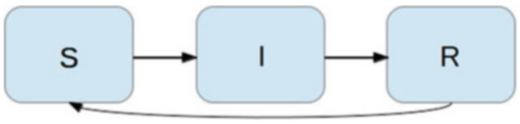
```
def demo_SIR():
    """Test case using a SIR model."""
    def f(u, t):
        S, I, R = u
        return [-beta*S*I, beta*S*I - gamma*I, gamma*I]

    beta = 10./(40*8*24)
    gamma = 3./(15*24)
    dt = 0.1             # 6 min
    D = 30               # Simulate for D days
    N_t = int(D*24/dt)   # Corresponding no of time steps
    T = dt*N_t           # End time
    U_0 = [50, 1, 0]

    u, t = ode_FE(f, U_0, dt, T)

    S = u[:,0]
    I = u[:,1]
    R = u[:,2]
    fig = plt.figure()
    l1, l2, l3 = plt.plot(t, S, t, I, t, R)
    fig.legend((l1, l2, l3), ('S', 'I', 'R'), 'center right')
    plt.xlabel('hours')
    plt.show()

    # Consistency check:
    N = S[0] + I[0] + R[0]
    eps = 1E-12  # Tolerance for comparing real numbers
    for n in range(len(S)):
        SIR_sum = S[n] + I[n] + R[n]
        if abs(SIR_sum - N) > eps:
            print('*** consistency check failed: S+I+R={:g} != {:g}'\
                  .format(SIR_sum, N))

if __name__ == '__main__':
    demo_SIR()
```

Recall that the `u` returned from `ode_FE` contains all components (S, I, R) in the solution vector at all time points. We therefore need to extract the S, I, and R values in separate arrays for further analysis and easy plotting.

Another key feature of this higher-quality code is the consistency check. By adding the three differential equations in the SIR model, we realize that $S' + I' + R' = 0$, which means that $S + I + R = \text{const}$. We can check that this relation holds by comparing $S^n + I^n + R^n$ to the sum of the initial conditions. Exercise 8.6 suggests another method for controlling the quality of the numerical solution.

8.3.7 Time-Restricted Immunity

Let us now assume that immunity after the disease only lasts for some certain time period. This means that there is transport from the R state to the S state:



Modeling the loss of immunity is very similar to modeling recovery from the disease: the amount of people losing immunity is proportional to the amount of recovered patients and the length of the time interval Δt. We can therefore write the loss in the R category as $-\nu \Delta t R$ in time Δt, where ν^{-1} is the typical time it takes to lose immunity. The loss in $R(t)$ is a gain in $S(t)$. The "budgets" for the categories therefore become

$$S^{n+1} = S^n - \beta \Delta t S^n I^n + \nu \Delta t R^n, \tag{8.30}$$

$$I^{n+1} = I^n + \beta \Delta t S^n I^n - \gamma \Delta t I^n, \tag{8.31}$$

$$R^{n+1} = R^n + \gamma \Delta t I^n - \nu \Delta t R^n\,. \tag{8.32}$$

Dividing by Δt and letting $\Delta t \to 0$ gives the differential equation system

$$S' = -\beta S I + \nu R, \tag{8.33}$$

$$I' = \beta S I - \gamma I, \tag{8.34}$$

$$R' = \gamma I - \nu R\,. \tag{8.35}$$

This system can be solved by the same methods as we demonstrated for the original SIR model. Only one modification in the program is necessary: adding `dt*nu*R[n]` to the `S[n+1]` update and subtracting the same quantity in the `R[n+1]` update:

```
for n in range(N_t):
    S[n+1] = S[n] - dt*beta*S[n]*I[n] + dt*nu*R[n]
    I[n+1] = I[n] + dt*beta*S[n]*I[n] - dt*gamma*I[n]
    R[n+1] = R[n] + dt*gamma*I[n] - dt*nu*R[n]
```

The modified code is found in the file `SIR2.py`.

Setting ν^{-1} to 50 days, reducing β by a factor of 4 compared to the previous example ($\beta = 0.00033$), and simulating for 300 days gives an oscillatory behavior in the categories, as depicted in Fig. 8.13. It is easy now to play around and study how the parameters affect the spreading of the disease. For example, making the disease slightly more effective (increase β to 0.00043) and increasing the average time to loss of immunity to 90 days lead to other oscillations, see Fig. 8.14.

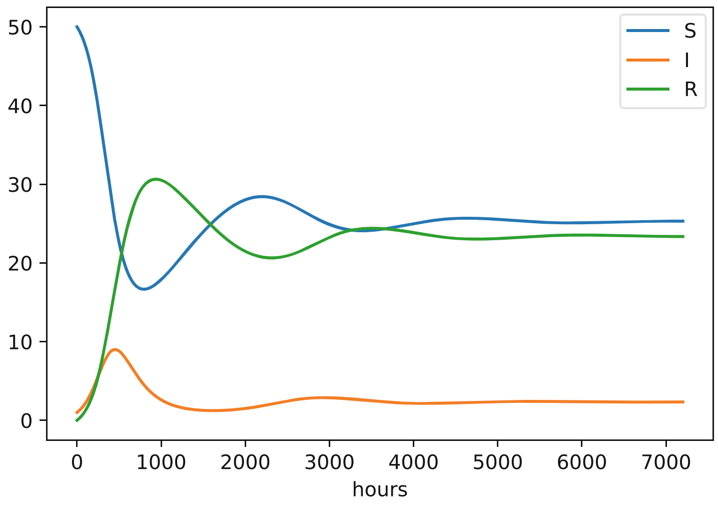


Fig. 8.13 Including loss of immunity

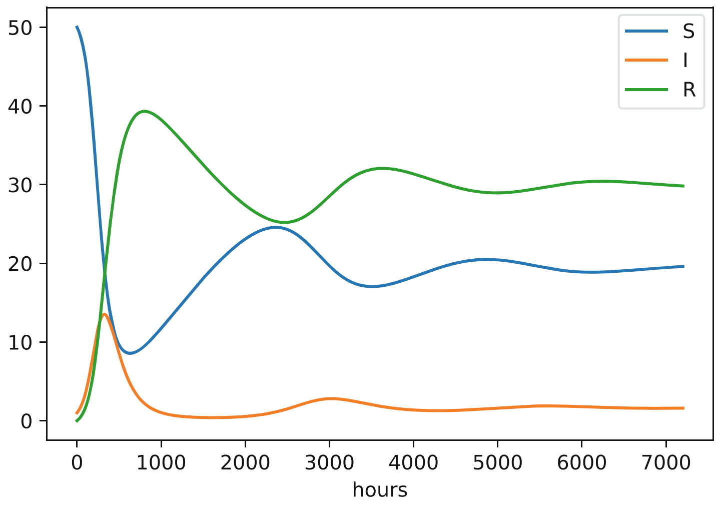
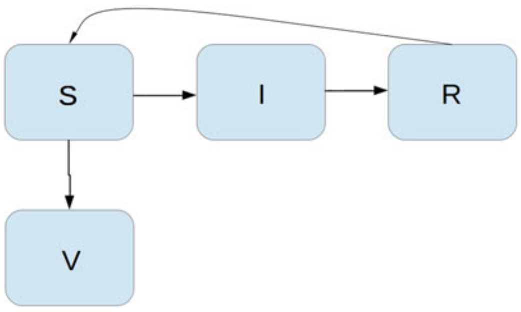


Fig. 8.14 Increasing β and reducing ν compared to Fig. 8.13

8.3.8 Incorporating Vaccination

We can extend the model to also include vaccination. To this end, it can be useful to track those who are vaccinated and those who are not. So, we introduce a fourth category, V, for those who have taken a successful vaccination. Furthermore, we assume that in a time interval Δt, a fraction $p\Delta t$ of the S category is subject to a successful vaccination. This means that in the time Δt, $p\Delta t S$ people leave from the S to the V category. Since the vaccinated ones cannot get the disease, there is no impact on the I or R categories. We can visualize the categories, and the movement between them, as



The new, extended differential equations with the V quantity become

$$S' = -\beta SI + \nu R - pS, \tag{8.36}$$

$$V' = pS, \tag{8.37}$$

$$I' = \beta SI - \gamma I, \tag{8.38}$$

$$R' = \gamma I - \nu R\,. \tag{8.39}$$

We shall refer to this model as the SIRV model.

The new equation for V' poses no difficulties when it comes to the numerical method. In a Forward Euler scheme we simply add an update

$$V^{n+1} = V^n + p\Delta t S^n\,.$$

The program needs to store $V(t)$ in an additional array `V`, and the plotting command must be extended with more arguments to plot `V` versus `t` as well. The complete code is found in the file `SIRV1.py`.

Using $p = 0.0005$ and $p = 0.0001$ as values for the vaccine efficiency parameter, the effect of vaccination is seen in Figs. 8.15 and 8.16, respectively. (other parameters are as in Fig. 8.13).

8.3.9 Discontinuous Coefficients: A Vaccination Campaign

What about modeling a vaccination campaign? Imagine that 6 days after the outbreak of the disease, the local health station launches a vaccination campaign. They reach out to many people, say 10 times as efficiently as in the previous (constant vaccination) case. If the campaign lasts for 10 days we can write

$$p(t) = \begin{cases} 0.005, & 6\cdot 24 \le t \le 15\cdot 24, \\ 0, & \text{otherwise} \end{cases}$$

Note that we must multiply the t value by 24 because t is measured in hours, not days. In the differential equation system, $pS(t)$ must be replaced by $p(t)S(t)$, and in this case we get a differential equation system with a term that is *discontinuous*. This

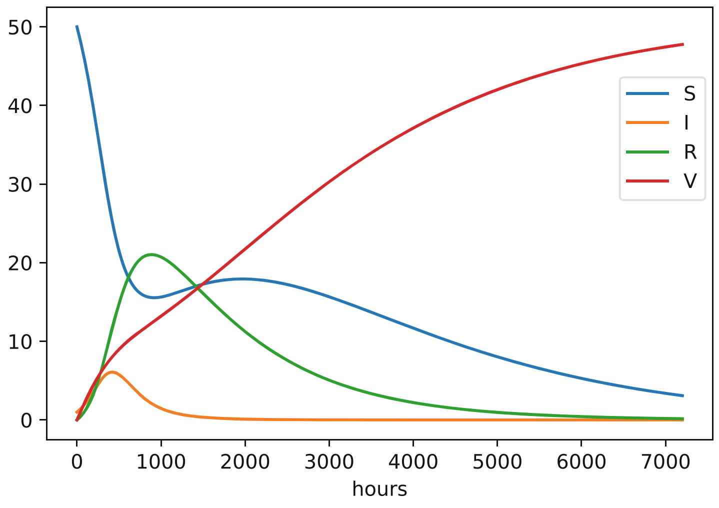


Fig. 8.15 The effect of vaccination: $p = 0.0005$

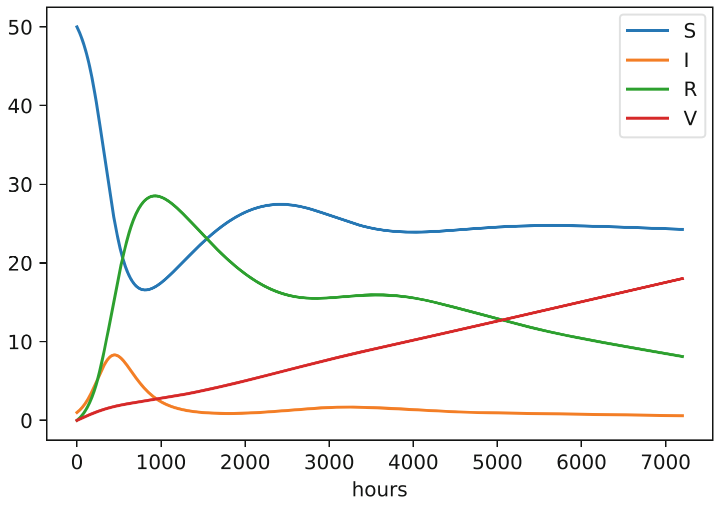


Fig. 8.16 The effect of vaccination: $p = 0.0001$

is usually quite a challenge in mathematics, but as long as we solve the equations numerically in a program, a discontinuous coefficient is easy to treat.

There are two ways to implement the discontinuous coefficient $p(t)$: through a function and through an array. The function approach is perhaps the easiest:

```
def p(t):
    return 0.005 if (6*24 <= t <= 15*24) else 0
```

Note the handy `if-else` construction in the return statement here. It is a one-line alternative to, for example,

```
if (6*24 <= t <= 15*24):
    return 0.005
```

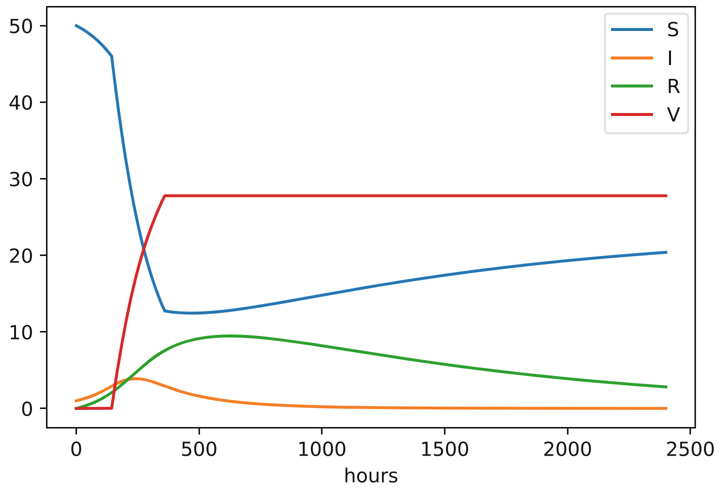


Fig. 8.17 The effect of a vaccination campaign

```
else:
    return 0
```

In the code for updating the arrays `S` and `V`, we then get a term `p(t[n])*S[n]`.

Alternatively, we can instead let $p(t)$ be an array filled with correct values prior to the simulation. Then we need to allocate an array `p` of length `N_t+1` and find the indices corresponding to the time period between 6 and 15 days. These indices are found from the time point divided by Δt. That is,

```
p = zeros(N_t+1)
start_index = 6*24/dt
stop_index = 15*24/dt
p[start_index:stop_index] = 0.005
```

The $p(t)S(t)$ term in the updating formulas for S and V simply becomes `p[n]*S[n]`. The file `SIRV2.py` contains a program based on filling an array `p`.

The effect of a vaccination campaign is illustrated in Fig. 8.17. All the data are as in Fig. 8.15, except that p is ten times stronger for a period of 10 days and $p = 0$ elsewhere.

8.4 Oscillating 1D Systems: A Second Order ODE

Numerous engineering constructions and devices contain materials that act like springs. Such springs give rise to oscillations, and controlling oscillations is a key engineering task. We shall now learn to simulate oscillating systems.

As always, we start with the simplest meaningful mathematical model, which for oscillations is a second-order differential equation:

$$u''(t) + \omega^2 u(t) = 0, \tag{8.40}$$

where ω is a given physical parameter. Equation (8.40) models a one-dimensional system oscillating without damping (i.e., with negligible damping). One-dimensional here means that some motion takes place along one dimension only in some coordinate system. Along with (8.40) we need the two *initial conditions* $u(0)$ and $u'(0)$.

8.4.1 Derivation of a Simple Model

Many engineering systems undergo oscillations, and differential equations constitute the key tool to understand, predict, and control the oscillations. We start with the simplest possible model that captures the essential dynamics of an oscillating system. Some body with mass m is attached to a spring and moves along a line without friction, see Fig. 8.18 for a sketch (rolling wheels indicate "no friction"). When the spring is stretched (or compressed), the spring force pulls (or pushes) the body back and work "against" the motion. More precisely, let $x(t)$ be the position of the body on the x axis, along which the body moves. The spring is not stretched when $x = 0$, so the force is zero, and $x = 0$ is hence the equilibrium position of the body. The spring force is $-kx$, where k is a constant to be measured. We assume that there are no other forces (e.g., no friction). Newton's second law of motion $F = ma$ then has $F = -kx$ and $a = \ddot{x}$,

$$-kx = m\ddot{x}, \tag{8.41}$$

which can be rewritten as

$$\ddot{x} + \omega^2 x = 0, \tag{8.42}$$

by introducing $\omega = \sqrt{k/m}$ (which is very common).

Equation (8.42) is a *second-order* differential equation, and therefore we need *two* initial conditions, one on the position $x(0)$ and one on the velocity $x'(0)$. Here

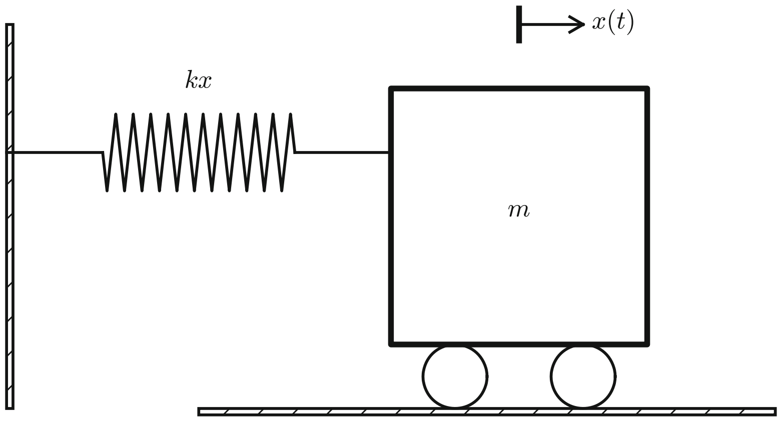


Fig. 8.18 Sketch of a one-dimensional, oscillating dynamic system (without friction)

we choose the body to be at rest, but moved away from its equilibrium position:

$$x(0) = X_0, \quad x'(0) = 0\,.$$

The exact solution of (8.42) with these initial conditions is $x(t) = X_0 \cos \omega t$. This can easily be verified by substituting into (8.42) and checking the initial conditions. The solution tells us that such a spring-mass system oscillates back and forth as described by a cosine curve.

The differential equation (8.42) appears in numerous other contexts. A classical example is a simple pendulum that oscillates back and forth. Physics books derive, from Newton's second law of motion, that

$$mL\theta'' + mg\sin\theta = 0,$$

where m is the mass of the body at the end of a pendulum with length L, g is the acceleration of gravity, and θ is the angle the pendulum makes with the vertical. Considering small angles θ, $\sin\theta \approx \theta$, and we get (8.42) with $x = \theta$, $\omega = \sqrt{g/L}$, $x(0) = \Theta$, and $x'(0) = 0$, if Θ is the initial angle and the pendulum is at rest at $t = 0$.

8.4.2 Numerical Solution

We have not looked at numerical methods for handling second-order derivatives, and such methods are an option, but we know how to solve first-order differential equations and even systems of first-order equations. With a little, yet very common, trick we can rewrite (8.42) as a first-order system of two differential equations. We introduce $u = x$ and $v = x' = u'$ as *two* new unknown functions. The two corresponding equations arise from the definition $v = u'$ and the original equation (8.42):

$$u' = v, \tag{8.43}$$

$$v' = -\omega^2 u\,. \tag{8.44}$$

(Notice that we can use $u'' = v'$ to remove the second-order derivative from Newton's second law.)

We can now apply the Forward Euler method to (8.43)–(8.44), exactly as we did in Sect. 8.3.2:

$$\frac{u^{n+1} - u^n}{\Delta t} = v^n, \tag{8.45}$$

$$\frac{v^{n+1} - v^n}{\Delta t} = -\omega^2 u^n, \tag{8.46}$$

resulting in the computational scheme

$$u^{n+1} = u^n + \Delta t\, v^n, \qquad (8.47)$$

$$v^{n+1} = v^n - \Delta t\, \omega^2 u^n\,. \qquad (8.48)$$

8.4.3 Programming the FE Scheme; the Special Case

A simple program for (8.47)–(8.48) follows the same ideas as in Sect. 8.3.3:

```
import numpy as np
import matplotlib.pyplot as plt

omega = 2
P = 2*np.pi/omega
dt = P/20
T = 3*P
N_t = int(round(T/dt))
t = np.linspace(0, N_t*dt, N_t+1)

u = np.zeros(N_t+1)
v = np.zeros(N_t+1)

# Initial condition
X_0 = 2
u[0] = X_0
v[0] = 0

# Step equations forward in time
for n in range(N_t):
    u[n+1] = u[n] + dt*v[n]
    v[n+1] = v[n] - dt*omega**2*u[n]

fig = plt.figure()
l1, l2 = plt.plot(t, u, 'b-', t, X_0*np.cos(omega*t), 'r--')
fig.legend((l1, l2), ('numerical', 'exact'), 'upper right')
plt.xlabel('t')
plt.savefig('tmp.pdf'); plt.savefig('tmp.png')
plt.show()
```

(See file `osc_FE.py`.)

Since we already know the exact solution as $u(t) = X_0 \cos \omega t$, we have reasoned as follows to find an appropriate simulation interval $[0, T]$ and also how many points we should choose. The solution has a period $P = 2\pi/\omega$. (The period P is the time difference between two peaks of the $u(t) \sim \cos \omega t$ curve.) Simulating for three periods of the cosine function, $T = 3P$, and choosing Δt such that there are 20 intervals per period gives $\Delta t = P/20$ and a total of $N_t = T/\Delta t$ intervals. The rest of the program is a straightforward coding of the Forward Euler scheme.

Figure 8.19 shows a comparison between the numerical solution and the exact solution of the differential equation. To our surprise, the numerical solution looks wrong. Is this discrepancy due to a programming error or a problem with the Forward Euler method?

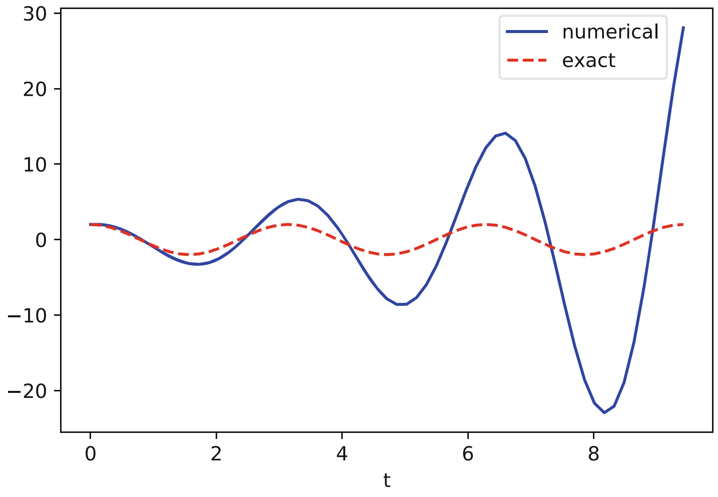


Fig. 8.19 Simulation of an oscillating system

First of all, even before trying to run the program, you should sit down and compute two steps in the time loop with a calculator so you have some intermediate results to compare with. Using $X_0 = 2$, $dt = 0.157079632679$, and $\omega = 2$, we get $u^1 = 2$, $v^1 = -1.25663706$, $u^2 = 1.80260791$, and $v^2 = -2.51327412$. Such calculations show that the program is seemingly correct. (Later, we can use such values to construct a unit test and a corresponding test function.)

The next step is to reduce the discretization parameter Δt and see if the results become more accurate. Figure 8.20 shows the numerical and exact solution for the cases $\Delta t = P/40$, $P/160$, $P/2000$. The results clearly become better, and the finest resolution gives graphs that cannot be visually distinguished. Nevertheless, the finest resolution involves 6000 computational intervals in total, which is considered quite much. This is no problem on a modern laptop, however, as the computations take just a fraction of a second.

Although 2000 intervals per oscillation period seem sufficient for an accurate numerical solution, the lower right graph in Fig. 8.20 shows that if we increase the simulation time, here to 20 periods, there is a little growth of the amplitude, which becomes significant over time. The conclusion is that the Forward Euler method has a fundamental problem with its growing amplitudes, and that a very small Δt is required to achieve satisfactory results. The longer the simulation is, the smaller Δt has to be. It is certainly time to look for more effective numerical methods!

8.4.4 A Magic Fix of the Numerical Method

In the Forward Euler scheme,

$$u^{n+1} = u^n + \Delta t\, v^n,$$

$$v^{n+1} = v^n - \Delta t\, \omega^2 u^n,$$

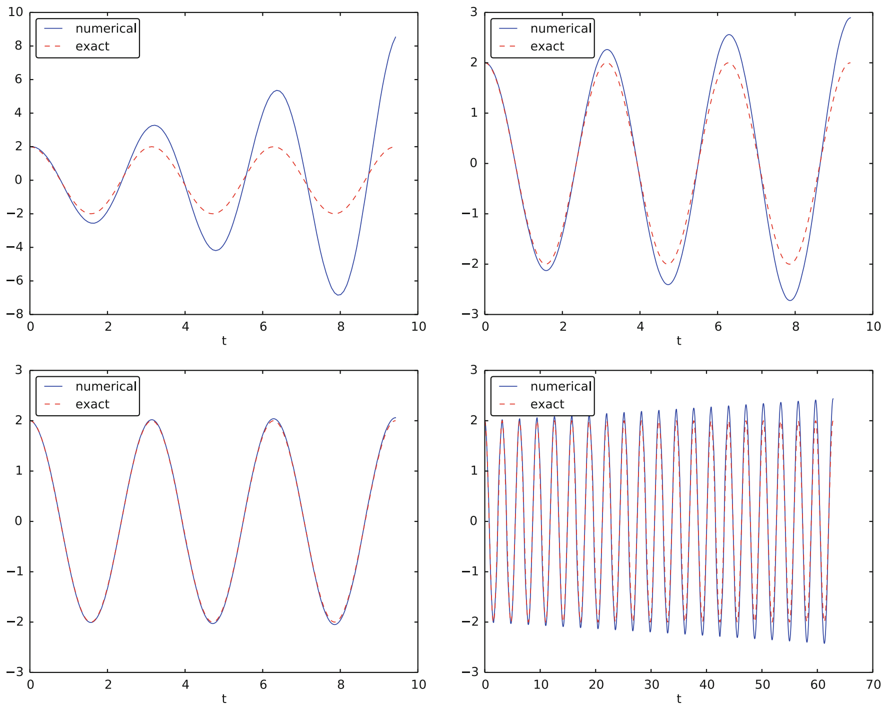


Fig. 8.20 Simulation of an oscillating system with different time steps. Upper left: 40 steps per oscillation period. Upper right: 160 steps per period. Lower left: 2000 steps per period. Lower right: 2000 steps per period, but longer simulation

we can replace u^n in the last equation by the recently computed value u^{n+1} from the first equation:

$$u^{n+1} = u^n + \Delta t\, v^n, \tag{8.49}$$

$$v^{n+1} = v^n - \Delta t\, \omega^2 u^{n+1}. \tag{8.50}$$

Before justifying this fix more mathematically, let us try it on the previous example. The results appear in Fig. 8.21. We see that the amplitude *does not grow*, but the phase is not entirely correct. After 40 periods (Fig. 8.21 right) we see a significant difference between the numerical and the exact solution. Decreasing Δt decreases the error. For example, with 2000 intervals per period, we only see a small phase error even after 50,000 periods (!). We can safely conclude that the fix results in an excellent numerical method!

Let us interpret the adjusted scheme mathematically. First we order (8.49)–(8.50) such that the difference approximations to derivatives become transparent:

$$\frac{u^{n+1} - u^n}{\Delta t} = v^n, \tag{8.51}$$

$$\frac{v^{n+1} - v^n}{\Delta t} = -\omega^2 u^{n+1}. \tag{8.52}$$

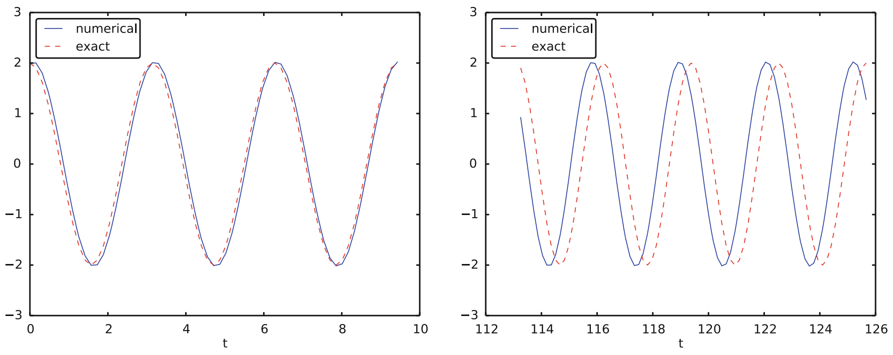


Fig. 8.21 Adjusted method: first three periods (left) and period 36–40 (right)

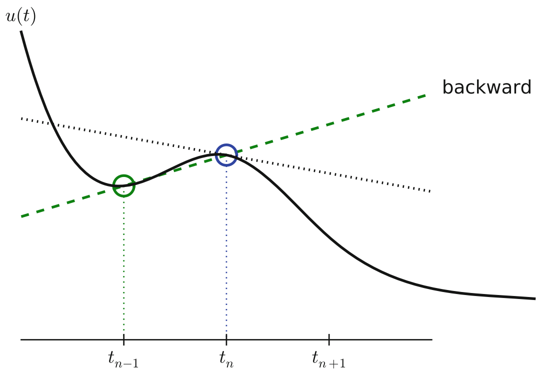


Fig. 8.22 Illustration of a backward difference approximation to the derivative

We interpret (8.51) as the differential equation sampled at mesh point t_n, because we have v^n on the right-hand side. The left-hand side is then a *forward difference* or Forward Euler approximation to the derivative u', see Fig. 8.4. On the other hand, we interpret (8.52) as the differential equation sampled at mesh point t_{n+1}, since we have u^{n+1} on the right-hand side. In this case, the difference approximation on the left-hand side is a *backward difference*,

$$v'(t_{n+1}) \approx \frac{v^{n+1} - v^n}{\Delta t} \quad \text{or} \quad v'(t_n) \approx \frac{v^n - v^{n-1}}{\Delta t} .$$

Figure 8.22 illustrates the backward difference. The error in the backward difference is proportional to Δt, the same as for the forward difference (but the proportionality constant in the error term has different sign). The resulting discretization method, seen in (8.52), is often referred to as a Backward Euler scheme (a first-order scheme, just like Forward Euler).

To summarize, using a forward difference for the first equation and a backward difference for the second equation results in a much better method than just using forward differences in both equations.

The standard way of expressing this scheme in physics is to change the order of the equations,

$$v' = -\omega^2 u, \tag{8.53}$$

$$u' = v, \tag{8.54}$$

and apply a forward difference to (8.53) and a backward difference to (8.54):

$$v^{n+1} = v^n - \Delta t\, \omega^2 u^n, \tag{8.55}$$

$$u^{n+1} = u^n + \Delta t\, v^{n+1}. \tag{8.56}$$

That is, first the velocity v is updated and then the position u, using the most recently computed velocity. There is no difference between (8.55)–(8.56) and (8.49)–(8.50) with respect to accuracy, so how you order the original differential equations does not matter. The scheme (8.55)–(8.56) goes by the name Semi-implicit Euler,[5] or Euler-Cromer (a first-order method). The implementation of (8.55)–(8.56) is found in the file `osc_EC.py`. The core of the code goes like

```python
u = zeros(N_t+1)
v = zeros(N_t+1)

# Initial condition
u[0] = 2
v[0] = 0

# Step equations forward in time
for n in range(N_t):
    v[n+1] = v[n] - dt*omega**2*u[n]
    u[n+1] = u[n] + dt*v[n+1]
```

Explicit and implicit methods

When we solve an ODE (linear or nonlinear) by the Forward Euler method, we get an explicit updating formula for the unknown at each time step, see, e.g., (8.6). Methods with this characteristic are known as *explicit*. We also have *implicit* methods. In that case, one or more algebraic equations must typically be solved for each time step. The Backward Euler method, for example, is such an implicit method (you will realize that when you do Exercise 8.24).

8.4.5 The Second-Order Runge-Kutta Method (or Heun's Method)

A very popular method for solving scalar and vector ODEs of first order is the second-order Runge-Kutta method (RK2), also known as Heun's method. The idea,

[5] http://en.wikipedia.org/wiki/Semi-implicit_Euler_method.

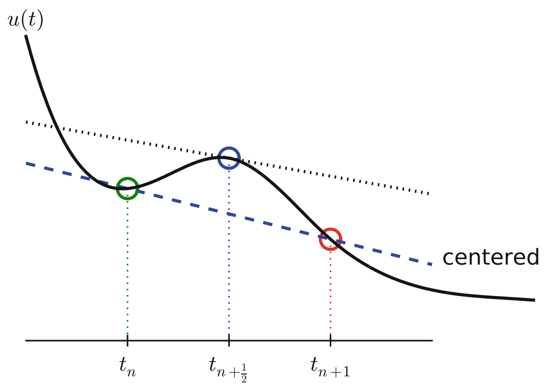


Fig. 8.23 Illustration of a centered difference approximation to the derivative

first thinking of a scalar ODE, is to form a *centered difference* approximation to the derivative between two time points:

$$u'(t_n + \frac{1}{2}\Delta t) \approx \frac{u^{n+1} - u^n}{\Delta t}.$$

The centered difference formula is visualized in Fig. 8.23. The error in the centered difference is proportional to Δt^2, one order higher than the forward and backward differences, which means that if we halve Δt, the error is more effectively reduced in the centered difference since it is reduced by a factor of four rather than two.

The problem with such a centered scheme for the general ODE $u' = f(u,t)$ is that we get

$$\frac{u^{n+1} - u^n}{\Delta t} = f(u^{n+\frac{1}{2}}, t_{n+\frac{1}{2}}),$$

which leads to difficulties since we do not know what $u^{n+\frac{1}{2}}$ is. However, we can approximate the value of f between two time levels by the arithmetic average of the values at t_n and t_{n+1}:

$$f(u^{n+\frac{1}{2}}, t_{n+\frac{1}{2}}) \approx \frac{1}{2}(f(u^n, t_n) + f(u^{n+1}, t_{n+1})).$$

This results in

$$\frac{u^{n+1} - u^n}{\Delta t} = \frac{1}{2}(f(u^n, t_n) + f(u^{n+1}, t_{n+1})),$$

which in general is a *nonlinear algebraic equation* for u^{n+1} if $f(u,t)$ is not a linear function of u. To deal with the unknown term $f(u^{n+1}, t_{n+1})$, without solving nonlinear equations, we can approximate or predict u^{n+1} using a Forward Euler step:

$$u^{n+1} = u^n + \Delta t f(u^n, t_n)\,.$$

This reasoning gives rise to the method

$$u^* = u^n + \Delta t f(u^n, t_n), \tag{8.57}$$

$$u^{n+1} = u^n + \frac{\Delta t}{2}(f(u^n, t_n) + f(u^*, t_{n+1}))\,. \tag{8.58}$$

The scheme applies to both scalar and vector ODEs.

For an oscillating system with $f = (v, -\omega^2 u)$ the file `osc_Heun.py` implements this method. The `demo` function in that file runs the simulation for 10 periods with 20 time steps per period. The corresponding numerical and exact solutions are shown in Fig. 8.24. We see that the amplitude grows, but not as much as for the Forward Euler method. However, the Euler-Cromer method performs better!

We should add that in problems where the Forward Euler method gives satisfactory approximations, such as growth/decay problems or the SIR model, the second-order Runge-Kutta method (Heun's method) usually works considerably better and produces greater accuracy for the same computational cost. It is therefore a very valuable method to be aware of, although it cannot compete with the Euler-Cromer scheme for oscillation problems. The derivation of the RK2/Heun scheme is also good general training in "numerical thinking".

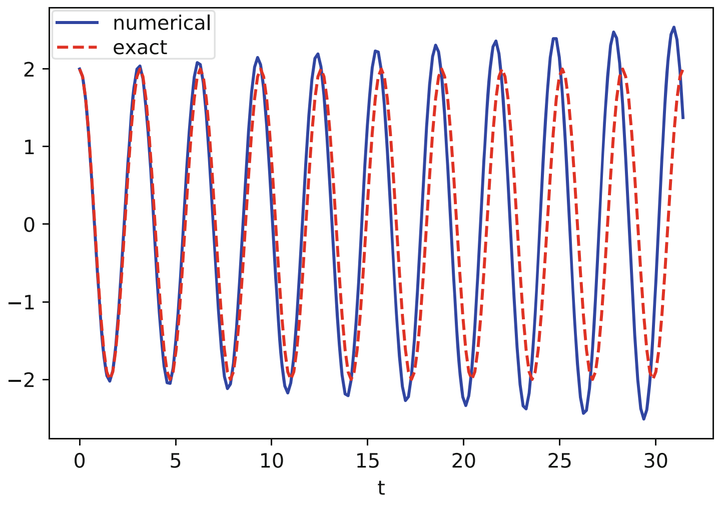


Fig. 8.24 Simulation of 10 periods of oscillations by Heun's method

8.4.6 Software for Solving ODEs

There is a jungle of methods for solving ODEs, and it would be nice to have easy access to implementations of a wide range of methods, especially the sophisticated and complicated *adaptive* methods that adjust Δt automatically to obtain a prescribed accuracy. The Python package `Odespy` (https://github.com/thomasantony/odespy/tree/py36/odespy) gives easy access to a lot of numerical methods for ODEs.

Odespy: Example with Exponential Growth The simplest possible example on using Odespy is to solve $u' = u$, $u(0) = 2$, for 100 time steps until $t = 4$:

```
import odespy
import numpy as np
import matplotlib.pyplot as plt

def f(u, t):
    return u

method = odespy.Heun      # or, e.g., odespy.ForwardEuler
solver = method(f)
solver.set_initial_condition(2)
time_points = np.linspace(0, 4, 101)
u, t = solver.solve(time_points)
plt.plot(t, u)
plt.show()
```

In other words, you define your right-hand side function `f(u, t)`, initialize an Odespy `solver` object, set the initial condition, compute a collection of time points where you want the solution, and ask for the solution. If you run the code, you get the expected plot of the exponential function (not shown).

A nice feature of Odespy is that problem parameters can be arguments to the user's `f(u, t)` function. For example, if our ODE problem is $u' = -au + b$, with two problem parameters a and b, we may write our `f` function as

```
def f(u, t, a, b):
    return -a*u + b
```

The extra, problem-dependent arguments `a` and `b` can be transferred to this function if we collect their values in a list or tuple when creating the Odespy solver and use the `f_args` argument:

```
a = 2
b = 1
solver = method(f, f_args=[a, b])
```

This is a good feature because problem parameters must otherwise be global variables—now they can be arguments in our right-hand side function in a natural way. Exercise 8.21 asks you to make a complete implementation of this problem and plot the solution.

Odespy: Example with Oscillations Using Odespy to solve oscillation ODEs like $u'' + \omega^2 u = 0$, reformulated as a system $u' = v$ and $v' = -\omega^2 u$, can be done with the following code:

```
import odespy
import numpy as np
import matplotlib.pyplot as plt

# Define the ODE system
# u' = v
# v' = -omega**2*u

def f(sol, t, omega=2):
    u, v = sol
    return [v, -omega**2*u]

# Set and compute problem dependent parameters
omega = 2
X_0 = 1
number_of_periods = 40
time_steps_per_period = 20
P = 2*np.pi/omega                    # length of one period
dt = P/time_steps_per_period         # time step
T = number_of_periods*P              # final simulation time

# Create Odespy solver object
odespy_method = odespy.RK2
solver = odespy_method(f, f_args=[omega])

# The initial condition for the system is collected in a list
solver.set_initial_condition([X_0, 0])

# Compute the desired time points where we want the solution
N_t = int(round(T/dt))               # no of time intervals
time_points = np.linspace(0, T, N_t+1)

# Solve the ODE problem
sol, t = solver.solve(time_points)

# Note: sol contains both displacement and velocity
# Extract original variables
u = sol[:,0]
v = sol[:,1]

plt.plot(t, u, t, v)  # ...for a quick check on u and v
plt.show()
```

After specifying the number of periods to simulate, as well as the number of time steps per period, we compute the time step (`dt`) and simulation end time (`T`).

The two statements `u = sol[:,0]` and `v = sol[:,1]` are important, since our two functions u and v in the ODE system are packed together in one array inside the Odespy solver (the solution of the ODE system is returned from `solver.solve` as a two-dimensional array where the first column (`sol[:,0]`) stores u and the second (`sol[:,1]`) stores v).

Remark

In the right-hand side function we write `f(sol, t, omega)` instead of `f(u, t, omega)` to indicate that the solution sent to `f` is a solution at time `t` where the values of u and v are packed together: `sol = [u, v]`. We might well use `u` as argument:

```
def f(u, t, omega=2):
    u, v = u
    return [v, -omega**2*u]
```

This just means that we redefine the name `u` inside the function to mean the solution at time `t` for the first component of the ODE system.

To switch to another numerical method, just substitute `RK2` by the proper name of the desired method. Typing `pydoc odespy` in the terminal window brings up a list of all the implemented methods. This very simple way of choosing a method suggests an obvious extension of the code above: we can define a list of methods, run all methods, and compare their u curves in a plot. As Odespy also contains the Euler-Cromer scheme, we rewrite the system with $v' = -\omega^2 u$ as the first ODE and $u' = v$ as the second ODE, because this is the standard choice when using the Euler-Cromer method (also in Odespy):

```
def f(u, t, omega=2):
    v, u = u
    return [-omega**2*u, v]
```

This change of equations also affects the initial condition: the first component is zero and second is `X_0`, so we need to pass the list `[0, X_0]` to `solver.set_initial_condition`.

The function `compare` in `osc_odespy.py` contains the details:

```
def compare(odespy_methods,
            omega,
            X_0,
            number_of_periods,
            time_intervals_per_period=20):

    P = 2*np.pi/omega                      # length of one period
    dt = P/time_intervals_per_period
    T = number_of_periods*P

    # If odespy_methods is not a list, but just the name of
    # a single Odespy solver, we wrap that name in a list
    # so we always have odespy_methods as a list
    if type(odespy_methods) != type([]):
        odespy_methods = [odespy_methods]

    # Make a list of solver objects
    solvers = [method(f, f_args=[omega]) for method in
               odespy_methods]
    for solver in solvers:
        solver.set_initial_condition([0, X_0])
    # Compute the time points where we want the solution
    N_t = int(round(T/dt))
    time_points = np.linspace(0, N_t*dt, N_t+1)

    legends = []
    for solver in solvers:
        sol, t = solver.solve(time_points)
        v = sol[:,0]
        u = sol[:,1]

        # Plot only the last p periods
        p = 6
        m = p*time_intervals_per_period  # no time steps to plot
        plt.plot(t[-m:], u[-m:])
        plt.hold('on')
        legends.append(solver.name())
        plt.xlabel('t')
    # Plot exact solution too
    plt.plot(t[-m:], X_0*np.cos(omega*t)[-m:], 'k--')
    legends.append('exact')
    plt.legend(legends, loc='lower left')
    plt.axis([t[-m], t[-1], -2*X_0, 2*X_0])
    plt.title('Simulation of {:d} periods with {:d} intervals per period'\
              .format(number_of_periods, time_intervals_per_period))
    plt.savefig('tmp.pdf'); plt.savefig('tmp.png')
    plt.show()
```

A new feature in this code is the ability to plot only the last p periods, which allows us to perform long time simulations and watch the end results without a cluttered plot with too many periods. The syntax t[-m:] plots the last m elements in t (a negative index in Python arrays/lists counts from the end).

We may compare Heun's method (i.e., the RK2 method) with the Euler-Cromer scheme:

```
compare(odespy_methods=[odespy.Heun, odespy.EulerCromer],
        omega=2, X_0=2, number_of_periods=20,
        time_intervals_per_period=20)
```

Figure 8.25 shows how Heun's method (blue line) has considerable error in both amplitude and phase already after 14–20 periods (upper left), but using three times as many time steps makes the curves almost equal (upper right). However, after 194–200 periods the errors have grown (lower left), but can be sufficiently reduced by halving the time step (lower right).

With all the methods in Odespy at hand, it is now easy to start exploring other methods, such as backward differences instead of the forward differences used in the Forward Euler scheme. Exercise 8.22 addresses that problem.

Odespy contains quite sophisticated adaptive methods where the user is "guaranteed" to get a solution with prescribed accuracy. There is no mathematical guarantee, but the error will for most cases not deviate significantly from the user's tolerance that reflects the accuracy. A very popular method of this type is the Runge-Kutta-Fehlberg method, which runs a fourth-order Runge-Kutta method and uses a fifth-order Runge-Kutta method to estimate the error so that Δt can be adjusted to keep the error below a tolerance. This method is also widely known as ode45, because that is the name of the function implementing the method in Matlab. We can

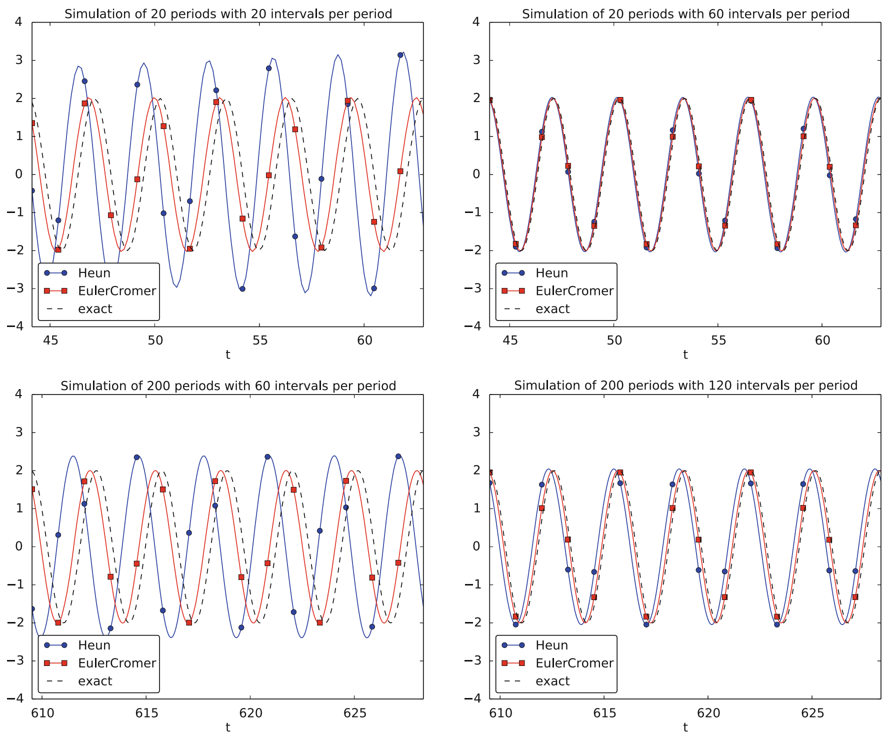


Fig. 8.25 Illustration of the impact of resolution (time steps per period) and length of simulation

easily test the Runge-Kutta-Fehlberg method as soon as we know the corresponding Odespy name, which is `RKFehlberg`:

```
compare(odespy_methods=[odespy.EulerCromer, odespy.RKFehlberg],
        omega=2, X_0=2, number_of_periods=200,
        time_intervals_per_period=40)
```

Note that the `time_intervals_per_period` argument refers to the time points where we want the solution. These points are also the ones used for numerical computations in the `odespy.EulerCromer` solver, while the `odespy.RKFehlberg` solver will use an unknown set of time points since the time intervals are adjusted as the method runs. One can easily look at the points actually used by the method as these are available as an array `solver.t_all` (but plotting or examining the points requires modifications inside the `compare` method).

Figure 8.26 shows a computational example where the Runge-Kutta-Fehlberg method is clearly superior to the Euler-Cromer scheme in long time simulations, but the comparison is not really fair because the Runge-Kutta-Fehlberg method applies about twice as many time steps in this computation and performs much more work per time step. It is quite a complicated task to compare two so different methods in a fair way so that the computational work versus accuracy is scientifically well reported.

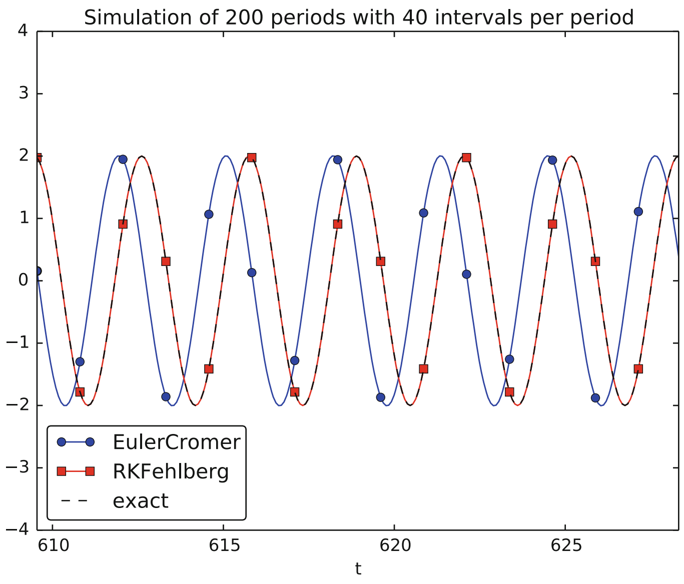


Fig. 8.26 Comparison of the Runge-Kutta-Fehlberg adaptive method against the Euler-Cromer scheme for a long time simulation (200 periods)

8.4.7 The Fourth-Order Runge-Kutta Method

The fourth-order Runge-Kutta method (RK4) is clearly the most widely used method to solve ODEs. Its power comes from high accuracy even with not so small time steps.

The Algorithm We first just state the four-stage algorithm:

$$u^{n+1} = u^n + \frac{\Delta t}{6}\left(f^n + 2\hat{f}^{n+\frac{1}{2}} + 2\tilde{f}^{n+\frac{1}{2}} + \bar{f}^{n+1}\right), \tag{8.59}$$

where

$$\hat{f}^{n+\frac{1}{2}} = f(u^n + \frac{1}{2}\Delta t f^n, t_{n+\frac{1}{2}}), \tag{8.60}$$

$$\tilde{f}^{n+\frac{1}{2}} = f(u^n + \frac{1}{2}\Delta t \hat{f}^{n+\frac{1}{2}}, t_{n+\frac{1}{2}}), \tag{8.61}$$

$$\bar{f}^{n+1} = f(u^n + \Delta t \tilde{f}^{n+\frac{1}{2}}, t_{n+1})\,. \tag{8.62}$$

Application We can run the same simulation as in Figs. 8.19, 8.21, and 8.24, for 40 periods. The 10 last periods are shown in Fig. 8.27. The results look as impressive as those of the Euler-Cromer method.

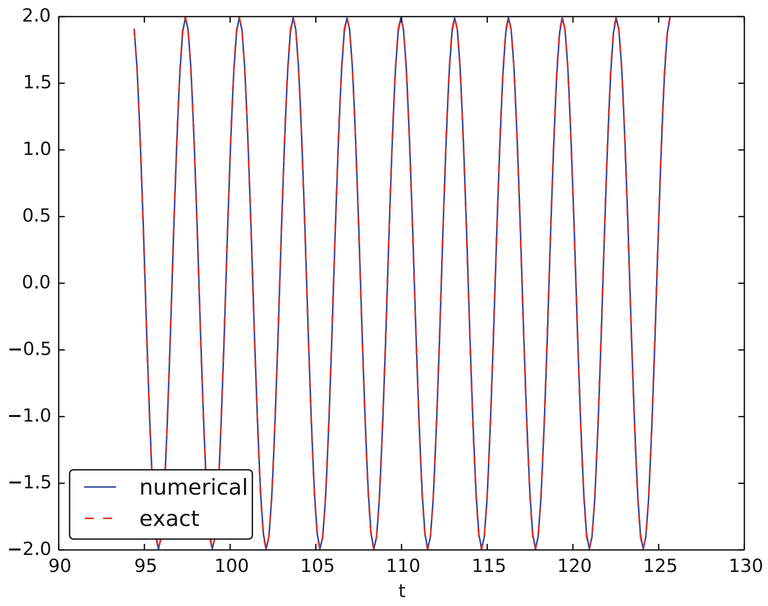


Fig. 8.27 The last 10 of 40 periods of oscillations by the fourth-order Runge-Kutta method

Implementation The stages in the fourth-order Runge-Kutta method can easily be implemented as a modification of the `osc_Heun.py` code. Alternatively, one can use the `osc_odespy.py` code by just providing the argument `odespy_methods=[odespy.RK4]` to the `compare` function.

Derivation The derivation of the fourth-order Runge-Kutta method can be presented in a pedagogical way that brings many fundamental elements of numerical discretization techniques together. It also illustrates many aspects of the "numerical thinking" required for constructing approximate solution methods.

We start with integrating the general ODE $u' = f(u,t)$ over a time step, from t_n to t_{n+1},

$$u(t_{n+1}) - u(t_n) = \int\limits_{t_n}^{t_{n+1}} f(u(t),t)dt\,.$$

The goal of the computation is $u(t_{n+1})$ (written u^{n+1}), while $u(t_n)$ (written u^n) is the most recently known value of u. The challenge with the integral is that the integrand involves the unknown u between t_n and t_{n+1}.

The integral can be approximated by the famous Simpson's rule[6]:

$$\int_{t_n}^{t_{n+1}} f(u(t),t)dt \approx \frac{\Delta t}{6}\left(f^n + 4f^{n+\frac{1}{2}} + f^{n+1}\right).$$

The problem with this formula is that we do not know $f^{n+\frac{1}{2}} = f(u^{n+\frac{1}{2}}, t_{n+\frac{1}{2}})$ and $f^{n+1} = (u^{n+1}, t_{n+1})$ as only u^n is available and only f^n can then readily be computed.

To proceed, the idea is to use various approximations for $f^{n+\frac{1}{2}}$ and f^{n+1} based on using well-known schemes for the ODE in the intervals $[t_n, t_{n+\frac{1}{2}}]$ and $[t_n, t_{n+1}]$. Let us split the integral into four terms:

$$\int_{t_n}^{t_{n+1}} f(u(t),t)dt \approx \frac{\Delta t}{6}\left(f^n + 2\hat{f}^{n+\frac{1}{2}} + 2\tilde{f}^{n+\frac{1}{2}} + \bar{f}^{n+1}\right),$$

where $\hat{f}^{n+\frac{1}{2}}$, $\tilde{f}^{n+\frac{1}{2}}$, and $\bar{f}^{n+1}$ are approximations to $f^{n+\frac{1}{2}}$ and f^{n+1} that can utilize already computed quantities. For $\hat{f}^{n+\frac{1}{2}}$ we can simply apply an approximation to $u^{n+\frac{1}{2}}$ based on a Forward Euler step of size $\frac{1}{2}\Delta t$:

$$\hat{f}^{n+\frac{1}{2}} = f(u^n + \frac{1}{2}\Delta t f^n, t_{n+\frac{1}{2}}) \tag{8.63}$$

This formula provides a prediction of $f^{n+\frac{1}{2}}$, so we can for $\tilde{f}^{n+\frac{1}{2}}$ try a Backward Euler method to approximate $u^{n+\frac{1}{2}}$:

$$\tilde{f}^{n+\frac{1}{2}} = f(u^n + \frac{1}{2}\Delta t \hat{f}^{n+\frac{1}{2}}, t_{n+\frac{1}{2}}). \tag{8.64}$$

With $\tilde{f}^{n+\frac{1}{2}}$ as an approximation to $f^{n+\frac{1}{2}}$, we can for the final term $\bar{f}^{n+1}$ use a midpoint method (or central difference, also called a Crank-Nicolson method) to approximate u^{n+1}:

$$\bar{f}^{n+1} = f(u^n + \Delta t \hat{f}^{n+\frac{1}{2}}, t_{n+1}). \tag{8.65}$$

We have now used the Forward and Backward Euler methods as well as the centered difference approximation in the context of Simpson's rule. The hope is that the combination of these methods yields an overall time-stepping scheme from t_n to t_n+1 that is much more accurate than the individual steps which have errors proportional to Δt and Δt^2. This is indeed true: the numerical error goes in fact like $C\Delta t^4$ for a constant C, which means that the error approaches zero very quickly as we reduce the time step size, compared to the Forward Euler method (error $\sim \Delta t$),

[6] http://en.wikipedia.org/wiki/Simpson's_rule.

the Euler-Cromer method (error $\sim \Delta t$) or the second-order Runge-Kutta, or Heun's, method (error $\sim \Delta t^2$).

Note that the fourth-order Runge-Kutta method is fully explicit so there is never any need to solve linear or nonlinear algebraic equations, regardless of what f looks like. However, the stability is conditional and depends on f. There is a large family of *implicit* Runge-Kutta methods that are unconditionally stable, but require solution of algebraic equations involving f at each time step. The Odespy package has support for a lot of sophisticated *explicit* Runge-Kutta methods, but not yet implicit Runge-Kutta methods.

8.4.8 More Effects: Damping, Nonlinearity, and External Forces

Our model problem $u'' + \omega^2 u = 0$ is the simplest possible mathematical model for oscillating systems. Nevertheless, this model makes strong demands to numerical methods, as we have seen, and is very useful as a benchmark for evaluating the performance of numerical methods.

Real-life applications involve more physical effects, which lead to a differential equation with more terms and also more complicated terms. Typically, one has a damping force $f(u')$ and a spring force $s(u)$. Both these forces may depend nonlinearly on their argument, u' or u. In addition, environmental forces $F(t)$ may act on the system. For example, the classical pendulum has a nonlinear "spring" or restoring force $s(u) \sim \sin(u)$, and air resistance on the pendulum leads to a damping force $f(u') \sim |u'|u'$. Examples on environmental forces include shaking of the ground (e.g., due to an earthquake) as well as forces from waves and wind.

With three types of forces on the system: F, f, and s, the sum of forces is written $F(t) - f(u') - s(u)$. Note the minus sign in front of f and s, which indicates that these functions are defined such that they represent forces acting *against* the motion. For example, springs attached to the wheels in a car are combined with effective dampers, each providing a damping force $f(u') = bu'$ that acts against the spring velocity u'. The corresponding physical force is then $-f$: $-bu'$, which points downwards when the spring is being stretched (and u' points upwards), while $-f$ acts upwards when the spring is being compressed (and u' points downwards).

Figure 8.28 shows an example of a mass m attached to a potentially nonlinear spring and dashpot, and subject to an environmental force $F(t)$. Nevertheless, our general model can equally well be a pendulum as in Fig. 8.29 with $s(u) = mg\sin\theta$ and $f(\dot{u}) = \frac{1}{2}C_D A\varrho\dot{\theta}|\dot{\theta}|$ (where $C_D = 0.4$, A is the cross sectional area of the body, and ϱ is the density of air).

Newton's second law for the system can be written with mass times acceleration on the left-hand side and the forces on the right-hand side:

$$mu'' = F(t) - f(u') - s(u)\,.$$

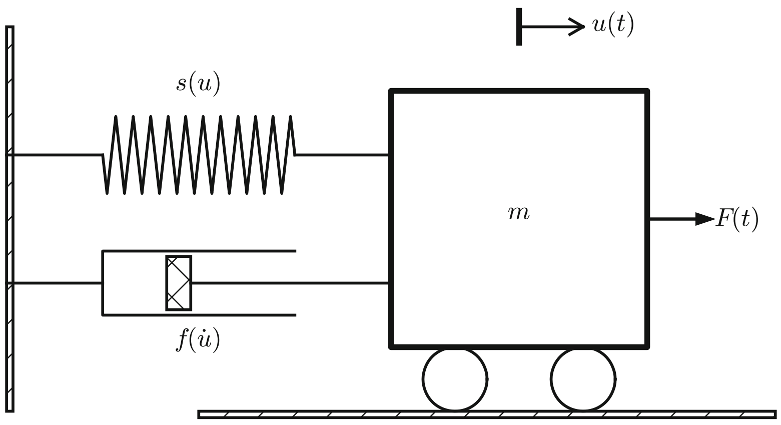


Fig. 8.28 General oscillating system

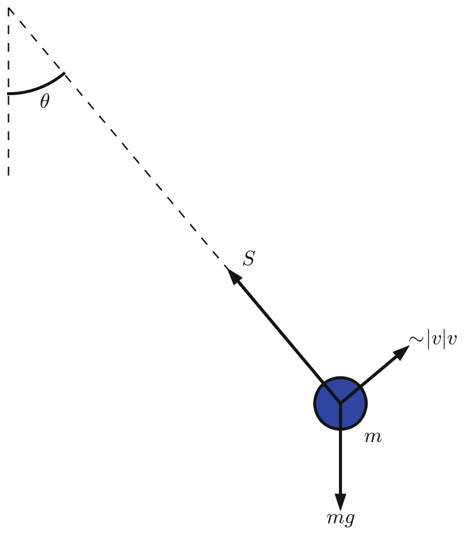


Fig. 8.29 A pendulum with forces

This equation is, however, more commonly reordered to

$$mu'' + f(u') + s(u) = F(t)\,. \tag{8.66}$$

Because the differential equation is of second order, due to the term u'', we need two initial conditions:

$$u(0) = U_0, \quad u'(0) = V_0\,. \tag{8.67}$$

Note that with the choices $f(u') = 0$, $s(u) = ku$, and $F(t) = 0$ we recover the original ODE $u'' + \omega^2 u = 0$ with $\omega = \sqrt{k/m}$.

How can we solve (8.66)? As for the simple ODE $u'' + \omega^2 u = 0$, we start by rewriting the second-order ODE as a system of two first-order ODEs:

$$v' = \frac{1}{m}\left(F(t) - s(u) - f(v)\right), \tag{8.68}$$

$$u' = v\,. \tag{8.69}$$

The initial conditions become $u(0) = U_0$ and $v(0) = V_0$.

Any method for a system of first-order ODEs can be used to solve for $u(t)$ and $v(t)$.

The Euler-Cromer Scheme An attractive choice from an implementational, accuracy, and efficiency point of view is the Euler-Cromer scheme where we take a forward difference in (8.68) and a backward difference in (8.69):

$$\frac{v^{n+1} - v^n}{\Delta t} = \frac{1}{m}\left(F(t_n) - s(u^n) - f(v^n)\right), \tag{8.70}$$

$$\frac{u^{n+1} - u^n}{\Delta t} = v^{n+1}, \tag{8.71}$$

We can easily solve for the new unknowns v^{n+1} and u^{n+1}:

$$v^{n+1} = v^n + \frac{\Delta t}{m}\left(F(t_n) - s(u^n) - f(v^n)\right), \tag{8.72}$$

$$u^{n+1} = u^n + \Delta t v^{n+1}\,. \tag{8.73}$$

Remark on the ordering of the ODEs

The ordering of the ODEs in the ODE system is important for the extended model (8.68)–(8.69). Imagine that we write the equation for u' first and then the one for v'. The Euler-Cromer method would then first use a forward difference for u^{n+1} and then a backward difference for v^{n+1}. The latter would lead to a *nonlinear* algebraic equation for v^{n+1},

$$v^{n+1} + \frac{\Delta t}{m} f(v^{n+1}) = v^n + \frac{\Delta t}{m}\left(F(t_{n+1}) - s(u^{n+1})\right),$$

if $f(v)$ is a nonlinear function of v. This would require a numerical method for nonlinear algebraic equations to find v^{n+1}, while updating v^{n+1} through a forward difference gives an equation for v^{n+1} that is linear and trivial to solve by hand.

The file osc_EC_general.py has a function EulerCromer that implements this method:

```
def EulerCromer(f, s, F, m, T, U_0, V_0, dt):
    import numpy as np
    N_t = int(round(T/dt))
    print('N_t:', N_t)
    t = np.linspace(0, N_t*dt, N_t+1)

    u = np.zeros(N_t+1)
    v = np.zeros(N_t+1)

    # Initial condition
    u[0] = U_0
    v[0] = V_0

    # Step equations forward in time
    for n in range(N_t):
        v[n+1] = v[n] + dt*(1./m)*(F(t[n]) - f(v[n]) - s(u[n]))
        u[n+1] = u[n] + dt*v[n+1]
    return u, v, t
```

The Fourth Order Runge-Kutta Method The RK4 method just evaluates the right-hand side of the ODE system,

$$(\frac{1}{m}\left(F(t) - s(u) - f(v)\right), v)$$

for known values of u, v, and t, so the method is very simple to use regardless of how the functions $s(u)$ and $f(v)$ are chosen.

8.4.9 Illustration of Linear Damping

We consider an engineering system with a linear spring, $s(u) = kx$, and a viscous damper, where the damping force is proportional to u', $f(u') = bu'$, for some constant $b > 0$. This choice may model the vertical spring system in a car (but engineers often like to illustrate such a system by a horizontally moving mass, like the one depicted in Fig. 8.28). We may choose simple values for the constants to illustrate basic effects of damping (and later excitations). Choosing the oscillations to be the simple $u(t) = \cos t$ function in the undamped case, we may set $m = 1$, $k = 1$, $b = 0.3$, $U_0 = 1$, $V_0 = 0$. The following function implements this case:

```
def linear_damping():
    import numpy as np
    b = 0.3
    f = lambda v: b*v
    s = lambda u: k*u
    F = lambda t: 0

    m = 1
    k = 1
    U_0 = 1
    V_0 = 0

    T = 12*np.pi
    dt = T/5000.

    u, v, t = EulerCromer(f=f, s=s, F=F, m=m, T=T,
                          U_0=U_0, V_0=V_0, dt=dt)
    plot_u(u, t)
```

The `plot_u` function is a collection of plot commands for plotting $u(t)$, or a part of it. Figure 8.30 shows the effect of the bu' term: we have oscillations with (an approximate) period 2π, as expected, but the amplitude is efficiently damped.

Remark about working with a scaled problem

Instead of setting $b = 0.3$ and $m = k = U_0 = 1$ as fairly "unlikely" physical values, it would be better to *scale* the equation $mu'' + bu' + ku = 0$. This means that we introduce dimensionless independent and dependent variables:

$$\bar{t} = \frac{t}{t_c}, \quad \bar{u} = \frac{u}{u_c},$$

where t_c and u_c are characteristic sizes of time and displacement, respectively, such that $\bar{t}$ and $\bar{u}$ have their typical size around unity (which minimizes rounding errors). In the present problem, we can choose $u_c = U_0$ and $t_c = \sqrt{m/k}$. This gives the following scaled (or dimensionless) problem for the dimensionless quantity $\bar{u}(\bar{t})$:

$$\frac{d^2\bar{u}}{d\bar{t}^2} + \beta\frac{d\bar{u}}{d\bar{t}} + \bar{u} = 0, \quad \bar{u}(0) = 1,\ \bar{u}'(0) = 0, \quad \beta = \frac{b}{\sqrt{mk}}\,.$$

The striking fact is that there is only *one* physical parameter in this problem: the dimensionless number β. Solving this problem corresponds to solving the original problem (with dimensions) with the parameters $m = k = U_0 = 1$ and $b = \beta$. However, solving the dimensionless problem is more general: if we have a solution $\bar{u}(\bar{t};\beta)$, we can find the physical solution of a range of problems since

$$u(t) = U_0\bar{u}(t\sqrt{k/m};\beta)\,.$$

As long as β is fixed, we can find u for any U_0, k, and m from the above formula! In this way, a time consuming simulation can be done only once, but still provide many solutions. This demonstrates the power of working with scaled or dimensionless problems.

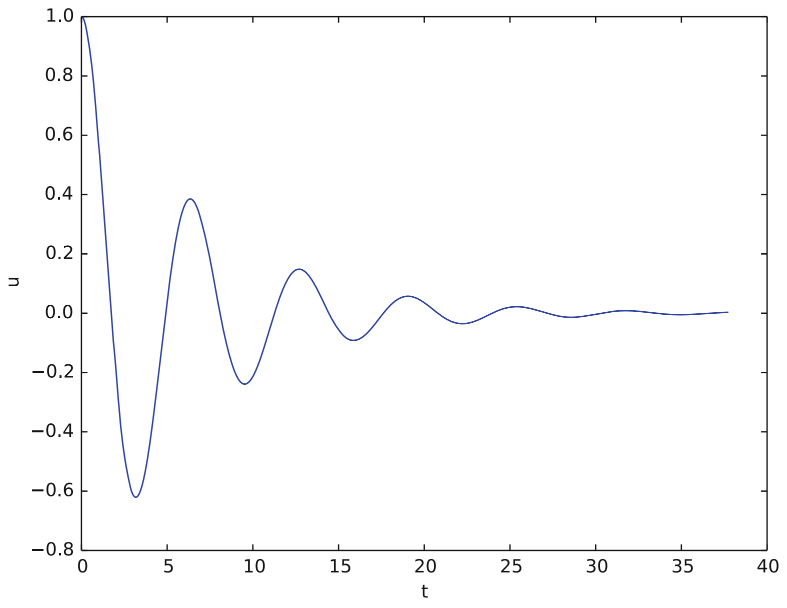


Fig. 8.30 Effect of linear damping

8.4.10 Illustration of Linear Damping with Sinusoidal Excitation

We now extend the previous example to also involve some external oscillating force on the system: $F(t) = A\sin(wt)$. Driving a car on a road with sinusoidal bumps might give such an external excitation on the spring system in the car (w is related to the velocity of the car).

With $A = 0.5$ and $w = 3$,

```
import math
w = 3
A = 0.5
F = lambda t: A*math.sin(w*t)
```

we get the graph in Fig. 8.31. The striking difference from Fig. 8.30 is that the oscillations start out as a damped $\cos t$ signal without much influence of the external force, but then the free oscillations of the undamped system ($\cos t$) $u'' + u = 0$ die out and the external force $0.5\sin(3t)$ induces oscillations with a shorter period $2\pi/3$. You are encouraged to play around with a larger A and switch from a sine to a cosine in F and observe the effects. If you look this up in a physics book, you can find exact analytical solutions to the differential equation problem in these cases.

A particularly interesting case arises when the excitation force has the same frequency as the free oscillations of the undamped system, i.e., $F(t) = A\sin t$. With the same amplitude $A = 0.5$, but a smaller damping $b = 0.1$, the oscillations in Fig. 8.31 becomes qualitatively very different as the amplitude grows significantly larger over some periods. This phenomenon is called *resonance* and is exemplified in Fig. 8.32. Removing the damping results in an amplitude that grows linearly in time.

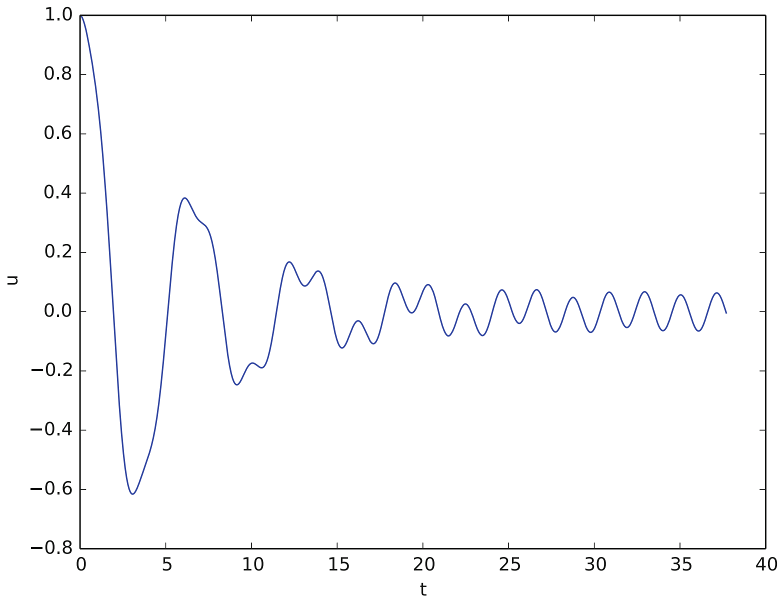


Fig. 8.31 Effect of linear damping in combination with a sinusoidal external force

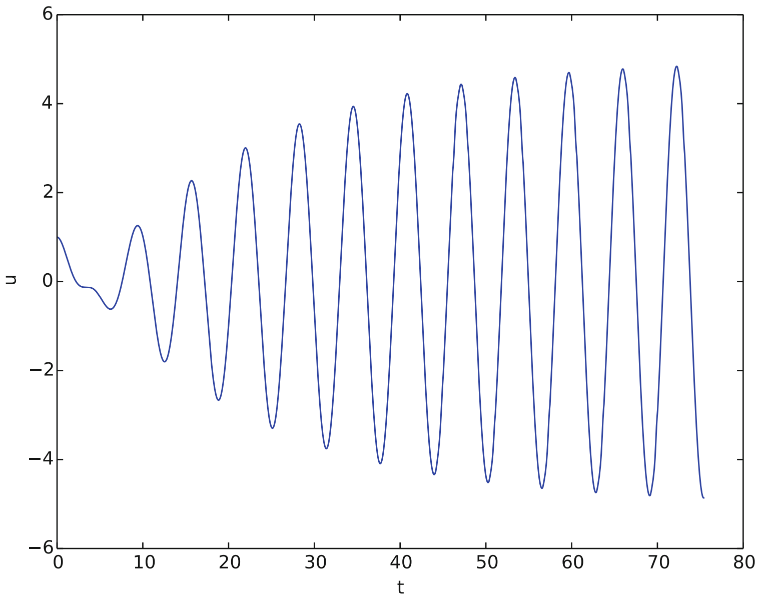


Fig. 8.32 Excitation force that causes resonance

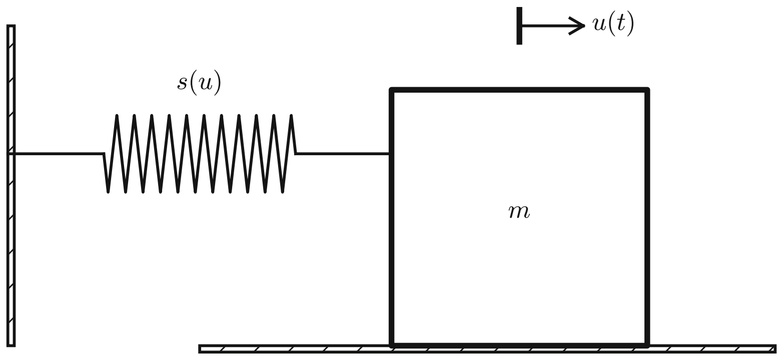


Fig. 8.33 Sketch of a one-dimensional, oscillating dynamic system subject to sliding friction and a spring force

8.4.11 Spring-Mass System with Sliding Friction

A body with mass m is attached to a spring with stiffness k while sliding on a plane surface. The body is also subject to a friction force $f(u')$ due to the contact between the body and the plane. Figure 8.33 depicts the situation. The friction force $f(u')$ can be modeled by Coulomb friction:

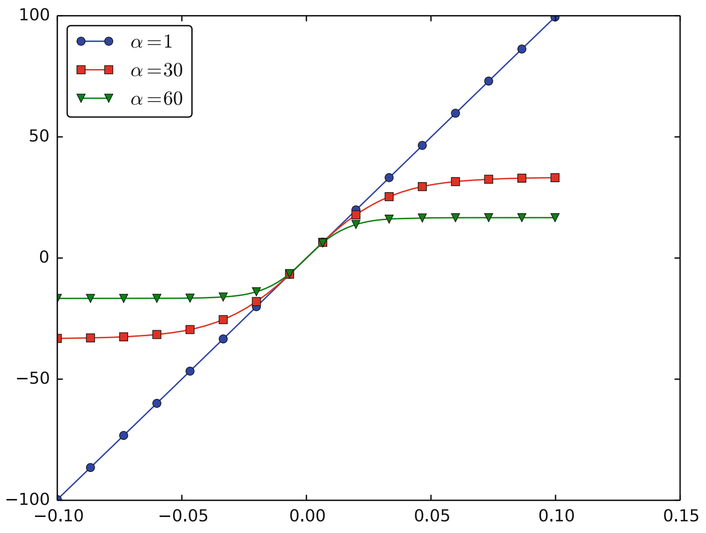
$$f(u') = \begin{cases} -\mu mg, & u' < 0, \\ \mu mg, & u' > 0, \\ 0, & u' = 0 \end{cases}$$

where μ is the friction coefficient, and mg is the normal force on the surface where the body slides. This formula can also be written as $f(u') = \mu mg\,\mathrm{sign}(u')$, provided the signum function $\mathrm{sign}(x)$ is defined to be zero for $x = 0$ (`numpy.sign` has this property). To check that the signs in the definition of f are right, recall that the actual physical force is $-f$ and this is positive (i.e., $f < 0$) when it works against the body moving with velocity $u' < 0$.

The nonlinear spring force is taken as

$$s(u) = -k\alpha^{-1}\tanh(\alpha u),$$

which is approximately $-ku$ for small u, but stabilizes at $\pm k/\alpha$ for large $\pm\alpha u$. Here is a plot with $k = 1000$ and $u \in [-0.1, 0.1]$ for three α values:



If there is no external excitation force acting on the body, we have the equation of motion

$$mu'' + \mu mg\,\mathrm{sign}(u') + k\alpha^{-1}\tanh(\alpha u) = 0\,.$$

Let us simulate a situation where a body of mass 1 kg slides on a surface with $\mu = 0.4$, while attached to a spring with stiffness $k = 1000$ kg/s^2. The initial displacement of the body is 10 cm, and the α parameter in $s(u)$ is set to 60 1/m. Using the `EulerCromer` function from the `osc_EC_general` code, we can write a function `sliding_friction` for solving this problem:

```
def sliding_friction():
    from numpy import tanh, sign

    f = lambda v: mu*m*g*sign(v)
    alpha = 60.0
    s = lambda u: k/alpha*tanh(alpha*u)
    F = lambda t: 0

    g = 9.81
    mu = 0.4
    m = 1
    k = 1000

    U_0 = 0.1
    V_0 = 0

    T = 2
    dt = T/5000.

    u, v, t = EulerCromer(f=f, s=s, F=F, m=m, T=T,
                          U_0=U_0, V_0=V_0, dt=dt)
    plot_u(u, t)
```

Running the `sliding_friction` function gives us the results in Fig. 8.34 with $s(u) = k\alpha^{-1}\tanh(\alpha u)$ (left) and the linearized version $s(u) = ku$ (right).

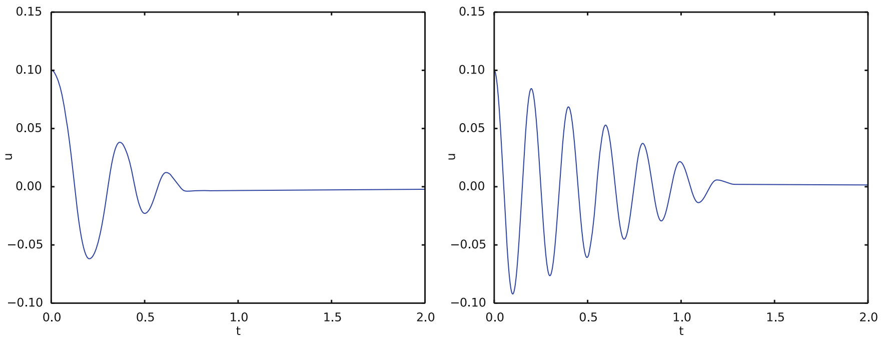


Fig. 8.34 Effect of nonlinear (left) and linear (right) spring on sliding friction

8.4.12 A Finite Difference Method; Undamped, Linear Case

We shall now address numerical methods for the second-order ODE

$$u'' + \omega^2 u = 0, \quad u(0) = U_0,\ u'(0) = 0,\ t \in (0, T],$$

without rewriting the ODE as a system of first-order ODEs. The primary motivation for "yet another solution method" is that the discretization principles result in a very good scheme, and more importantly, the thinking around the discretization can be reused when solving partial differential equations.

The main idea of this numerical method is to approximate the second-order derivative u'' by a finite difference. While there are several choices of difference approximations to first-order derivatives, there is one dominating formula for the second-order derivative:

$$u''(t_n) \approx \frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2}\,. \tag{8.74}$$

The error in this approximation is proportional to Δt^2. Letting the ODE be valid at some arbitrary time point t_n,

$$u''(t_n) + \omega^2 u(t_n) = 0,$$

we just insert the approximation (8.74) to get

$$\frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2} = -\omega^2 u^n\,. \tag{8.75}$$

We now assume that u^{n-1} and u^n are already computed and that u^{n+1} is the new unknown. Solving with respect to u^{n+1} gives

$$u^{n+1} = 2u^n - u^{n-1} - \Delta t^2\omega^2 u^n\,. \tag{8.76}$$

A major problem arises when we want to start the scheme. We know that $u^0 = U_0$, but applying (8.76) for $n = 0$ to compute u^1 leads to

$$u^1 = 2u^0 - u^{-1} - \Delta t^2 \omega^2 u^0, \tag{8.77}$$

where we do not know u^{-1}. The initial condition $u'(0) = 0$ can help us to eliminate u^{-1}—and this condition must anyway be incorporated in some way. To this end, we discretize $u'(0) = 0$ by a *centered difference*,

$$u'(0) \approx \frac{u^1 - u^{-1}}{2\Delta t} = 0\,.$$

It follows that $u^{-1} = u^1$, and we can use this relation to eliminate u^{-1} in (8.77):

$$u^1 = u^0 - \frac{1}{2}\Delta t^2 \omega^2 u^0\,. \tag{8.78}$$

With $u^0 = U_0$ and u^1 computed from (8.78), we can compute u^2, u^3, and so forth from (8.76). Exercise 8.25 asks you to explore how the steps above are modified in case we have a nonzero initial condition $u'(0) = V_0$.

Remark on a simpler method for computing u^1

We could approximate the initial condition $u'(0)$ by a forward difference:

$$u'(0) \approx \frac{u^1 - u^0}{\Delta t} = 0,$$

leading to $u^1 = u^0$. Then we can use (8.76) for the coming time steps. However, this forward difference has an error proportional to Δt, while the centered difference we used has an error proportional to Δt^2, which is compatible with the accuracy (error goes like Δt^2) used in the discretization of the differential equation.

The method for the second-order ODE described above goes under the name Störmer's method or Verlet integration.[7] It turns out that this method is mathematically equivalent with the Euler-Cromer scheme (!). Or more precisely, the general formula (8.76) is equivalent with the Euler-Cromer formula, but the scheme for the first time level (8.78) implements the initial condition $u'(0)$ slightly more accurately than what is naturally done in the Euler-Cromer scheme. The latter will do

$$v^1 = v^0 - \Delta t \omega^2 u^0, \quad u^1 = u^0 + \Delta t v^1 = u^0 - \Delta t^2 \omega^2 u^0,$$

[7] http://en.wikipedia.org/wiki/Verlet_integration.

which differs from u^1 in (8.78) by an amount $\frac{1}{2}\Delta t^2\omega^2 u^0$.

Because of the equivalence of (8.76) with the Euler-Cromer scheme, the numerical results will have the same nice properties such as a constant amplitude. There will be a phase error as in the Euler-Cromer scheme, but this error is effectively reduced by reducing Δt, as already demonstrated.

The implementation of (8.78) and (8.76) is straightforward in a function (file `osc_2nd_order.py`):

```
import numpy as np

def osc_2nd_order(U_0, omega, dt, T):
    """
    Solve u'' + omega**2*u = 0 for t in (0,T], u(0)=U_0 and u'(0)=0,
    by a central finite difference method with time step dt.
    """
    Nt = int(round(T/dt))
    u = np.zeros(Nt+1)
    t = np.linspace(0, Nt*dt, Nt+1)

    u[0] = U_0
    u[1] = u[0] - 0.5*dt**2*omega**2*u[0]
    for n in range(1, Nt):
        u[n+1] = 2*u[n] - u[n-1] - dt**2*omega**2*u[n]
    return u, t
```

8.4.13 A Finite Difference Method; Linear Damping

A key issue is how to generalize the scheme from Sect. 8.4.12 to a differential equation with more terms. We start with the case of a linear damping term $f(u') = bu'$, a possibly nonlinear spring force $s(u)$, and an excitation force $F(t)$:

$$mu'' + bu' + s(u) = F(t), \quad u(0) = U_0,\ u'(0) = 0,\ t \in (0, T]\,. \tag{8.79}$$

We need to find the appropriate difference approximation to u' in the bu' term. A good choice is the *centered difference*

$$u'(t_n) \approx \frac{u^{n+1} - u^{n-1}}{2\Delta t}\,. \tag{8.80}$$

Sampling the equation at a time point t_n,

$$mu''(t_n) + bu'(t_n) + s(u^n) = F(t_n),$$

and inserting the finite difference approximations to u'' and u' results in

$$m\frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2} + b\frac{u^{n+1} - u^{n-1}}{2\Delta t} + s(u^n) = F^n, \tag{8.81}$$

where F^n is a short notation for $F(t_n)$. Equation (8.81) is linear in the unknown u^{n+1}, so we can easily solve for this quantity:

$$u^{n+1} = (2mu^n + (\frac{b}{2}\Delta t - m)u^{n-1} + \Delta t^2(F^n - s(u^n)))(m + \frac{b}{2}\Delta t)^{-1}\,. \quad (8.82)$$

As in the case without damping, we need to derive a special formula for u^1. The initial condition $u'(0) = 0$ implies also now that $u^{-1} = u^1$, and with (8.82) for $n = 0$, we get

$$u^1 = u^0 + \frac{\Delta t^2}{2m}(F^0 - s(u^0))\,. \quad (8.83)$$

In the more general case with a nonlinear damping term $f(u')$,

$$mu'' + f(u') + s(u) = F(t),$$

we get

$$m\frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2} + f(\frac{u^{n+1} - u^{n-1}}{2\Delta t}) + s(u^n) = F^n,$$

which is a *nonlinear algebraic equation* for u^{n+1} that must be solved by numerical methods. A much more convenient scheme arises from using a backward difference for u',

$$u'(t_n) \approx \frac{u^n - u^{n-1}}{\Delta t},$$

because the damping term will then be known, involving only u^n and u^{n-1}, and we can easily solve for u^{n+1}.

The downside of the backward difference compared to the centered difference (8.80) is that it reduces the order of the accuracy in the overall scheme from Δt^2 to Δt. In fact, the Euler-Cromer scheme evaluates a nonlinear damping term as $f(v^n)$ when computing v^{n+1}, and this is equivalent to using the backward difference above. Consequently, the convenience of the Euler-Cromer scheme for nonlinear damping comes at a cost of lowering the overall accuracy of the scheme from second to first order in Δt. Using the same trick in the finite difference scheme for the second-order differential equation, i.e., using the backward difference in $f(u')$, makes this scheme equally convenient and accurate as the Euler-Cromer scheme in the general nonlinear case $mu'' + f(u') + s(u) = F$.

8.5 Rate of Convergence

In this chapter, we have seen how the numerical solutions improve as the time step Δt is reduced, just like we would expect. Thinking back on numerical computation of integrals (Chap. 6), we experienced the same when reducing the sub-interval size h, i.e., computations became more accurate. Not too surprising then, the asymptotic error models are similar, and the convergence rate is computed in essentially the same way (except that the error computation requires some more consideration with the methods of the present chapter). Let us look at the details.

8.5.1 Asymptotic Behavior of the Error

For numerical methods that solve ODEs, it is known that when $\Delta t \rightarrow 0$, the approximation error[8] usually behaves like

$$E = C\,(\Delta t)^r\ , \tag{8.84}$$

for positive constants C and r. The constant r is known as the *convergence rate*, and its value will depend on the method (r could be 1, 2 or 4, for example). A method with convergence rate r is said to be an r-th order method, and we understand that the larger the r value, the quicker the error E drops when the time step Δt is reduced.

8.5.2 Computing the Convergence Rate

Consider a set of experiments, $i = 0, 1, \ldots$, each depending on a discretization parameter Δt_i that typically is halved from one experiment to the next. For each experiment, a corresponding error E_i is computed. We may then estimate r (C is not really interesting) from two experiments:

$$\begin{aligned} E_{i-1} &= C\Delta t_{i-1}^r \\ E_i &= C\Delta t_i^r\ . \end{aligned}$$

We eliminate C by, e.g., dividing the latter equation by the former, and proceed to solve for r:

$$r = \frac{\ln(E_i/E_{i-1})}{\ln(\Delta t_i/\Delta t_{i-1})}\ .$$

Clearly, r will vary with the pair of experiments used in the above formula, i.e., the value of i. Thus, what we actually compute, is a sequence of r_{i-1} values ($i = 1, 2, \ldots$), where each r_{i-1} value is computed from two experiments $(E_i, \Delta t_i)$ and $(E_{i-1}, \Delta t_{i-1})$. Since the error model is asymptotic (i.e., valid as $\Delta t \rightarrow 0$), the r value corresponding to the smallest Δt value will be the best estimate of the convergence rate.

But How Do We Compute the Error E_i? When we previously addressed the computing of convergence rates for numerical integration methods (trapezoidal and

[8] As will be addressed below, there are several options for how to quantify this error.

midpoint methods), the error was a single number for each choice of n, i.e., the number of sub-intervals. With each Δt_i now, however, there is basically one error *for each point in the mesh* (remember: we compute an approximation to a *function* now, not a single number, as we did with integrals)! What we would like to have, is a single number E_i that we could refer to as "the error" of the corresponding experiment.

There are different ways to arrive at E_i, and we might reason as follows. For each of our experiments, at each point in the mesh, there will be a difference (generally not zero) between the true solution and our computed approximation. The collection of all these differences makes up an error mesh function, having values at the mesh points only. Also, with a total of N_t time steps, there will be $N_t + 1$ points in the time mesh (N_t increases when Δt decreases, for a fixed time interval $[0, T]$). Thus, for each Δt_i, the error mesh function comprises $N_t + 1$ function values e^n, $n = 0, 1, \ldots, N_t$. Now, one simple way to get to E_i, is to use the maximum of all e^n values,[9] comparing absolute values. In fact, we did that previously in `test_ode_FE_exact_linear.py` (Sect. 8.2.7).

Other alternatives utilize a constructed error function $e(t)$, being a *continuous* function of time. The function $e(t)$ may be generated from the error mesh function by simply connecting successive e^n values with straight lines. With $e(t)$ in place, one may choose to use the L^2 *norm* (read "L-two norm") of $e(t)$,

$$\|e\|_{L^2} = \sqrt{\int_0^T e(t)^2 dt}\,, \tag{8.85}$$

for E_i. The L^2 norm has nice mathematical properties and is much used. When computing the integral of the L^2 norm, we may use the trapezoidal method, and let the integration points coincide with the mesh points. Because of the straight lines composing $e(t)$, the integral computation will then become exact (within machine precision). Thus, assuming a uniform mesh, we can proceed to write (8.85) as

$$\|e\|_{L^2} \approx \sqrt{\Delta t \left(\frac{1}{2}(e_0) + \frac{1}{2}\left(e_{N_t}\right) + \sum_{n=1}^{N_t-1} (e_n)^2 \right)}\,, \tag{8.86}$$

Finally, we make yet another approximation, by simply disregarding the contributions from e_0 and e_{N_t}. This is acceptable, since these contributions go to zero as $\Delta t \to 0$. The resulting expression is called the *discrete* L^2 norm, and is denoted by l^2. In this way, we get the final and simpler expression[10] as

$$\|e_n\|_{l^2} = \sqrt{\Delta t \sum_{n=1}^{N_t-1} (e_n)^2}\,. \tag{8.87}$$

[9] This is referred to as the discrete (L^∞) norm (read "L-infinity norm", but often called the "max norm") for the error mesh function e^n.

[10] We should add that, in this expression, Δt may be switched with $\frac{T}{N_t}$, followed by dropping T, since this common scaling factor is independent of the vector values. Finally, it is usually preferred to use the length of the vector, i.e. $N_t + 1$ in stead of N_t.

With (8.87), we have a simple way of computing the error E_i, letting

$$E_i = \|e_n\|_{l^2} \ . \tag{8.88}$$

We are now in position to compute convergence rates, and write corresponding test functions, also for ODE solvers.

8.5.3 Test Function: Convergence Rates for the FE Solver

To illustrate, we write a simple test function for `ode_FE` that we implemented previously (Sect. 8.2.5). Applying the solver to a population growth model, the test function could be written:

```python
def test_convergence_rates_ode_FE(number_of_experiments):
    """
    Test that the convergence rate with the ode_FE solver is 1.
    Use population growth model as test case.
    """
    U_0=100      # initial value
    T=20         # total time span
    dt = 2.0     # initial time step
    expected_rate_FE = 1.0

    def f(u, t):
        """Population growth, u' = a*u, with a = 0.1 here."""
        return 0.1*u
    def u_exact(t):
        return 100*np.exp(0.1*t)

    dt_values = []
    E_values = []
    for i in range(number_of_experiments):
        u, t = ode_FE(f=f, U_0=U_0, dt=dt, T=T)
        u_e = u_exact(t)  # get exact solution at time mesh points (in t)
        E = np.sqrt(dt*np.sum((u_e-u)**2))   # ...discrete L^2 norm

        dt_values.append(dt)
        E_values.append(E)
        dt = dt/2               # Halving time step for next solve

    r = [np.log(E_values[i]/E_values[i-1])/
         np.log(dt_values[i]/dt_values[i-1])
         for i in range(1, number_of_experiments, 1)]
    #print(r)

    # Accept rate to 1 decimal place
    tol = 0.1
    assert abs(r[-1] - expected_rate_FE) < tol
    return
```

When `test_convergence_rates_ode_FE` is called, the `for` loop will execute `ode_FE` the number of times specified by the input parameter `number_of_experiments`. Each execution of `ode_FE` happens with half the time step of the previous execution. Errors (`E`) and time steps (`dt`) are stored in corresponding lists, so that convergence rates (`r`) can be computed after the loop. Observe, in the loop, how `ode_FE` returns (the solution `u` and) the time mesh `t`, which then is used as input to the `u_exact` function, causing the exact function values to also be computed *at* the very same mesh points as `u`.

The Forward Euler method is a first order method, so we should get r near 1 as the time step becomes small enough. A call to this test function does indeed confirm (remove # in front of `print(r)`) that `r` comes very close to 1 as `dt` gets smaller.

8.6 Exercises

Exercise 8.1: Restructure a Given Code
Section 8.1.1 gives a code for computing the development of water volume V in a tank. Restructure the code by introducing an appropriate function `compute_V` that computes and returns the volumes, and a function `application` that calls the former function and plots the result.

Note that your restructuring should not cause any change in program behavior, as experienced by a user of the program.
Filename: `restruct_tank_case1.py`.

Exercise 8.2: Geometric Construction of the Forward Euler Method
Section 8.2.4 describes a geometric interpretation of the Forward Euler method. This exercise will demonstrate the geometric construction of the solution in detail. Consider the differential equation $u' = u$ with $u(0) = 1$. We use time steps $\Delta t = 1$.

a) Start at $t = 0$ and draw a straight line with slope $u'(0) = u(0) = 1$. Go one time step forward to $t = \Delta t$ and mark the solution point on the line.
b) Draw a straight line through the solution point $(\Delta t, u^1)$ with slope $u'(\Delta t) = u^1$. Go one time step forward to $t = 2\Delta t$ and mark the solution point on the line.
c) Draw a straight line through the solution point $(2\Delta t, u^2)$ with slope $u'(2\Delta t) = u^2$. Go one time step forward to $t = 3\Delta t$ and mark the solution point on the line.
d) Set up the Forward Euler scheme for the problem $u' = u$. Calculate u^1, u^2, and u^3. Check that the numbers are the same as obtained in a)-c).

Filename: `ForwardEuler_geometric_solution.py`.

Exercise 8.3: Make Test Functions for the Forward Euler Method
The purpose of this exercise is to make a file `test_ode_FE.py` that makes use of the `ode_FE` function in the file `ode_FE.py` and automatically verifies the implementation of `ode_FE`.

a) The solution computed by hand in Exercise 8.2 can be used as a reference solution. Make a function `test_ode_FE_1()` that calls `ode_FE` to compute three time steps in the problem $u' = u$, $u(0) = 1$, and compare the three values u^1, u^2, and u^3 with the values obtained in Exercise 8.2.
b) The test in a) can be made more general using the fact that if f is linear in u and does not depend on t, i.e., we have $u' = ru$, for some constant r, the Forward Euler method has a closed form solution as outlined in Sect. 8.2.1: $u^n = U_0(1 + r\Delta t)^n$. Use this result to construct a test function `test_ode_FE_2()` that

runs a number of steps in `ode_FE` and compares the computed solution with the listed formula for u^n.

Filename: `test_ode_FE.py`.

Exercise 8.4: Implement and Evaluate Heun's Method

a) A second-order Runge-Kutta method, also known has Heun's method, is derived in Sect. 8.4.5. Make a function `ode_Heun(f, U_0, dt, T)` (as a counterpart to `ode_FE(f, U_0, dt, T)` in `ode_FE.py`) for solving a scalar ODE problem $u' = f(u, t)$, $u(0) = U_0$, $t \in (0, T]$, with this method using a time step size Δt.
b) Solve the simple ODE problem $u' = u$, $u(0) = 1$, by the `ode_Heun` and the `ode_FE` function. Make a plot that compares Heun's method and the Forward Euler method with the exact solution $u(t) = e^t$ for $t \in [0, 6]$. Use a time step $\Delta t = 0.5$.
c) For the case in b), find through experimentation the largest value of Δt where the exact solution and the numerical solution by Heun's method cannot be distinguished visually. It is of interest to see how far off the curve the Forward Euler method is when Heun's method can be regarded as "exact" (for visual purposes).

Filename: `ode_Heun.py`.

Exercise 8.5: Find an Appropriate Time Step; Logistic Model

Compute the numerical solution of the logistic equation for a set of repeatedly halved time steps: $\Delta t_k = 2^{-k}\Delta t$, $k = 0, 1, \ldots$. Plot the solutions corresponding to the last two time steps Δt_k and Δt_{k-1} in the same plot. Continue doing this until you cannot visually distinguish the two curves in the plot. Then one has found a sufficiently small time step.

Hint Extend the `logistic.py` file. Introduce a loop over k, write out Δt_k, and ask the user if the loop is to be continued.

Filename: `logistic_dt.py`.

Exercise 8.6: Find an Appropriate Time Step; SIR Model

Repeat Exercise 8.5 for the SIR model.

Hint Import the `ode_FE` function from the `ode_system_FE` module and make a modified `demo_SIR` function that has a loop over repeatedly halved time steps. Plot S, I, and R versus time for the two last time step sizes in the same plot.

Filename: `SIR_dt.py`.

Exercise 8.7: Model an Adaptive Vaccination Campaign

In the SIRV model with time-dependent vaccination from Sect. 8.3.9, we want to test the effect of an adaptive vaccination campaign where vaccination is offered as long as half of the population is not vaccinated. The campaign starts after Δ days. That is, $p = p_0$ if $V < \frac{1}{2}(S^0 + I^0)$ and $t > \Delta$ days, otherwise $p = 0$.

Demonstrate the effect of this vaccination policy: choose β, γ, and ν as in Sect. 8.3.9, set $p = 0.001$, $\Delta = 10$ days, and simulate for 200 days.

Hint This discontinuous $p(t)$ function is easiest implemented as a Python function containing the indicated `if` test. You may use the file `SIRV1.py` as starting point, but note that it implements a time-dependent $p(t)$ via an array.

Filename: `SIRV_p_adapt.py`.

Exercise 8.8: Make a SIRV Model with Time-Limited Effect of Vaccination
We consider the SIRV model from Sect. 8.3.8, but now the effect of vaccination is time-limited. After a characteristic period of time, π, the vaccination is no more effective and individuals are consequently moved from the V to the S category and can be infected again. Mathematically, this can be modeled as an average leakage $-\pi^{-1}V$ from the V category to the S category (i.e., a gain $\pi^{-1}V$ in the latter). Write up the complete model, implement it, and rerun the case from Sect. 8.3.8 with various choices of parameters to illustrate various effects.
Filename: `SIRV1_V2S.py`.

Exercise 8.9: Refactor a Flat Program
Consider the file `osc_FE.py` implementing the Forward Euler method for the oscillating system model (8.43)–(8.44). The `osc_FE.py` code is what we often refer to as a flat program, meaning that it is just one main program with no functions. Your task is to *refactor* the code in `osc_FE.py` according to the specifications below. Refactoring, means to alter the inner structure of the code, while, to a user, the program works just as before.

To easily reuse the numerical computations in other contexts, place the part that produces the numerical solution (allocation of arrays, initializing the arrays at time zero, and the time loop) in a function `osc_FE(X_0, omega, dt, T)`, which returns `u, v, t`. Place the particular computational example in `osc_FE.py` in a function `demo()`. Construct the file `osc_FE_func.py` such that the `osc_FE` function can easily be reused in other programs. In Python, this means that `osc_FE_func.py` is a module that can be imported in other programs. The requirement of a module is that there should be no main program, except in the test block. You must therefore call `demo` from a test block (i.e., the block after `if __name__ == '__main__'`).
Filename: `osc_FE_func.py`.

Exercise 8.10: Simulate Oscillations by a General ODE Solver
Solve the system (8.43)–(8.44) using the general solver `ode_FE` described in Sect. 8.3.6. Program the ODE system and the call to the `ode_FE` function in a separate file `osc_ode_FE.py`.

Equip this file with a test function that reads a file with correct u values and compares these with those computed by the `ode_FE` function. To find correct u

values, modify the program `osc_FE.py` to dump the `u` array to file, run `osc_FE.py`, and let the test function read the reference results from that file.
Filename: `osc_ode_FE.py`.

Exercise 8.11: Compute the Energy in Oscillations

a) Make a function `osc_energy(u, v, omega)` for returning the potential and kinetic energy of an oscillating system described by (8.43)–(8.44). The potential energy is taken as $\frac{1}{2}\omega^2 u^2$ while the kinetic energy is $\frac{1}{2}v^2$. (Note that these expressions are not exactly the *physical* potential and kinetic energy, since these would be $\frac{1}{2}mv^2$ and $\frac{1}{2}ku^2$ for a model $mx'' + kx = 0$.)

   Place the `osc_energy` in a separate file `osc_energy.py` such that the function can be called from other functions.
b) Add a call to `osc_energy` in the programs `osc_FE.py` and `osc_EC.py` and plot the *sum* of the kinetic and potential energy. How does the total energy develop for the Forward Euler and the Euler-Cromer schemes?

Filenames: `osc_energy.py`, `osc_FE_energy.py`, `osc_EC_energy.py`.

Exercise 8.12: Use a Backward Euler Scheme for Population Growth

We consider the ODE problem $N'(t) = rN(t)$, $N(0) = N_0$. At some time, $t_n = n\Delta t$, we can approximate the derivative $N'(t_n)$ by a *backward difference*, see Fig. 8.22:

$$N'(t_n) \approx \frac{N(t_n) - N(t_n - \Delta t)}{\Delta t} = \frac{N^n - N^{n-1}}{\Delta t},$$

which leads to

$$\frac{N^n - N^{n-1}}{\Delta t} = rN^n\,,$$

called the *Backward Euler scheme*.

a) Find an expression for the N^n in terms of N^{n-1} and formulate an algorithm for computing N^n, $n = 1, 2, \ldots, N_t$.
b) Implement the algorithm in a) in a function `growth_BE(N_0, dt, T)` for solving $N' = rN$, $N(0) = N_0$, $t \in (0, T]$, with time step Δt (`dt`).
c) Implement the Forward Euler scheme in a function `growth_FE(N_0, dt, T)` as described in b).
d) Compare visually the solution produced by the Forward and Backward Euler schemes with the exact solution when $r = 1$ and $T = 6$. Make two plots, one with $\Delta t = 0.5$ and one with $\Delta t = 0.05$.

Filename: `growth_BE.py`.

Exercise 8.13: Use a Crank-Nicolson Scheme for Population Growth
It is recommended to do Exercise 8.12 prior to the present one. Here we look at the same population growth model $N'(t) = rN(t)$, $N(0) = N_0$. The time derivative $N'(t)$ can be approximated by various types of finite differences. Exercise 8.12 considers a backward difference (Fig. 8.22), while Sect. 8.2.2 explained the forward difference (Fig. 8.4). A *centered difference* is more accurate than a backward or forward difference:

$$N'(t_n + \frac{1}{2}\Delta t) \approx \frac{N(t_n + \Delta t) - N(t_n)}{\Delta t} = \frac{N^{n+1} - N^n}{\Delta t} .$$

This type of difference, applied at the point $t_{n+\frac{1}{2}} = t_n + \frac{1}{2}\Delta t$, is illustrated geometrically in Fig. 8.23.

a) Insert the finite difference approximation in the ODE $N' = rN$ and solve for the unknown N^{n+1}, assuming N^n is already computed and hence known. The resulting computational scheme is often referred to as a *Crank-Nicolson* scheme.
b) Implement the algorithm in a) in a function `growth_CN(N_0, dt, T)` for solving $N' = rN$, $N(0) = N_0$, $t \in (0, T]$, with time step Δt (`dt`).
c) Make plots for comparing the Crank-Nicolson scheme with the Forward and Backward Euler schemes in the same test problem as in Exercise 8.12.

Filename: `growth_CN.py`.

Exercise 8.14: Understand Finite Differences via Taylor Series
The Taylor series around a point $x = a$ can for a function $f(x)$ be written

$$\begin{aligned} f(x) &= f(a) + \frac{d}{dx}f(a)(x-a) + \frac{1}{2!}\frac{d^2}{dx^2}f(a)(x-a)^2 \\ &\quad + \frac{1}{3!}\frac{d^3}{dx^3}f(a)(x-a)^3 + \ldots \\ &= \sum_{i=0}^{\infty}\frac{1}{i!}\frac{d^i}{dx^i}f(a)(x-a)^i . \end{aligned}$$

For a function of time, as addressed in our ODE problems, we would use u instead of f, t instead of x, and a time point t_n instead of a:

$$\begin{aligned} u(t) &= u(t_n) + \frac{d}{dt}u(t_n)(t-t_n) + \frac{1}{2!}\frac{d^2}{dt^2}u(t_n)(t-t_n)^2 \\ &\quad + \frac{1}{3!}\frac{d^3}{dt^3}u(t_n)(t-t_n)^3 + \ldots \\ &= \sum_{i=0}^{\infty}\frac{1}{i!}\frac{d^i}{dt^i}u(t_n)(t-t_n)^i . \end{aligned}$$

a) A forward finite difference approximation to the derivative $f'(a)$ reads

$$u'(t_n) \approx \frac{u(t_n + \Delta t) - u(t_n)}{\Delta t}\,.$$

We can justify this formula mathematically through Taylor series. Write up the Taylor series for $u(t_n + \Delta t)$ (around $t = t_n$, as given above), and then solve the expression with respect to $u'(t_n)$. Identify, on the right-hand side, the finite difference approximation *and* an infinite series. This series is then the error in the finite difference approximation. If Δt is assumed small (i.e. $\Delta t << 1$), Δt will be much larger than Δt^2, which will be much larger than Δt^3, and so on. The *leading order term* in the series for the error, i.e., the error with the least power of Δt is a good approximation of the error. Identify this term.

b) Repeat a) for a backward difference:

$$u'(t_n) \approx \frac{u(t_n) - u(t_n - \Delta t)}{\Delta t}\,.$$

This time, write up the Taylor series for $u(t_n - \Delta t)$ around t_n. Solve with respect to $u'(t_n)$, and identify the leading order term in the error. How is the error compared to the forward difference?

c) A centered difference approximation to the derivative, as explored in Exercise 8.13, can be written

$$u'(t_n + \frac{1}{2}\Delta t) \approx \frac{u(t_n + \Delta t) - u(t_n)}{\Delta t}\,.$$

Write up the Taylor series for $u(t_n)$ around $t_n + \frac{1}{2}\Delta t$ and the Taylor series for $u(t_n + \Delta t)$ around $t_n + \frac{1}{2}\Delta t$. Subtract the two series, solve with respect to $u'(t_n + \frac{1}{2}\Delta t)$, identify the finite difference approximation and the error terms on the right-hand side, and write up the leading order error term. How is this term compared to the ones for the forward and backward differences?

d) Can you use the leading order error terms in a)–c) to explain the visual observations in the numerical experiment in Exercise 8.13?

e) Find the leading order error term in the following standard finite difference approximation to the second-order derivative:

$$u''(t_n) \approx \frac{u(t_n + \Delta t) - 2u(t_n) + u(t_n - \Delta t)}{\Delta t^2}\,.$$

Hint Express $u(t_n \pm \Delta t)$ via Taylor series and insert them in the difference formula.

Filename: `Taylor_differences.pdf`.

Exercise 8.15: The Leapfrog Method

We consider the general ODE problem $u'(t) = f(u, t)$, $u(0) = U_0$. To solve such an ODE numerically, the second order *Leapfrog method* approximates the derivative (at some time $t_n = n\Delta t$) by use of a *centered difference* over two time steps,

$$u'(t_n) \approx \frac{u(t_{n+1}) - u(t_{n-1})}{2\Delta t} = \frac{u^{n+1} - u^{n-1}}{2\Delta t}.$$

a) Replace the derivative in the ODE by the given centered difference approximation and show that this allows us to formulate:

$$u^{n+1} = u^{n-1} + 2\Delta t f(u^n, t_n)\,, \qquad n = 1, 2, \ldots, N_t - 1,$$

with $u^0 = U_0$. Do we have the information we need to get the scheme started?

b) The problem you discovered in the previous question, may be fixed by using the Forward Euler method. However, the Leapfrog method is a second order method, while the Forward Euler method is first order.

Argue, with reference to the Taylor series (see, e.g., Exercise 8.14), why the Forward Euler method can be used without reducing the order of the overall scheme.

c) Implement the Leapfrog scheme in a function `leapfrog`. Make sure the function takes an appropriate set of input parameters, so that it is easy to import and use.

d) Write a function `compare_FE_leapfrog` that compares graphically the solutions produced by the Forward Euler and Leapfrog methods, when they solve the population growth model $u' = 0.1u$, with $u(0) = 100$. Let the total time span $T = 20$, and use a time step $dt = 2$. In the plot produced, include also the exact solution, so that the numerical solutions can be assessed.

e) Suggest a reasonable asymptotic error model before you write a proper test function `test_convergence_rates` that may be used to compute and check the convergence rates of the implemented Leapfrog method. However, the test function should take appropriate input parameters, so that it can be used also for other ODE solvers, in particular the `ode_FE` implemented previously.

Include your test function in a program, together with the two functions you defined previously (`leapfrog` and `compare_FE_leapfrog`). Write the code with a test block, so that it gets easy to either import functions from the module, or to run it as a program.

Finally, run the program (so that `compare_FE_leapfrog` gets called, as well as `test_convergence_rates` for both FE and Leapfrog) and confirm that it works as expected. In particular, does the plot look good, and do you get the convergence rates you expected for Forward Euler and Leapfrog?

Filename: `growth_leapfrog.py`.

Exercise 8.16: The Runge-Kutta Third Order Method
A general ODE problem $u'(t) = f(u,t)$, $u(0) = U_0$, may be solved numerically by the third order Runge-Kutta method. The computational scheme reads

$$\begin{aligned}
u^{n+1} &= u^n + \frac{\Delta t}{6}\left(k_1 + 4k_2 + k_3\right), \qquad n = 0, 1, \ldots, N_t - 1,\\
k_1 &= f(u^n, t_n),\\
k_2 &= f(u^n + \frac{\Delta t}{2}, t_n + \frac{\Delta t}{2}),\\
k_3 &= f(u^n - \Delta t k_1 + 2\Delta t k_2, t_n + \Delta t),
\end{aligned}$$

with $u^0 = U_0$.

a) Implement the scheme in a function `RK3` that takes appropriate parameters, so that it is easy to import and use whenever needed.
b) Write a function `compare_FE_RK3` that compares graphically the solutions produced by the Forward Euler and RK3 methods, when they solve the population growth model $u' = 0.1u$, with $u(0) = 100$. Let the total time span $T = 20$, and use a time step $dt = 2$. In the plot produced, include also the exact solution, so that the numerical solutions can be assessed.
c) Suggest a reasonable asymptotic error model before you write a proper test function `test_convergence_rates` that may be used to compute and check the convergence rates of the implemented RK3 method. However, the test function should take appropriate input parameters, so that it can be used also for other ODE solvers, in particular the `ode_FE` implemented previously (if you already have written this test function when doing Exercise 8.15, you may prefer to import the function).

   Include your test function in a program, together with the two functions you defined previously (`RK3` and `compare_FE_RK3`). Write the code with a test block, so that it gets easy to either import functions from the module, or to run it as a program.

   Finally, run the program (so that `compare_FE_RK3` gets called, as well as `test_convergence_rates` for both FE and RK3) and confirm that it works as expected. In particular, does the plot look good, and do you get the convergence rates you expected for Forward Euler and RK3?

Filename: `runge_kutta_3.py`.

Exercise 8.17: The Two-Step Adams-Bashforth Method
Differing from the *single-step* methods presented in this chapter, we have the *multi-step* methods, for example the Adams-Bashforth methods. With the single-step methods, u^{n+1} is computed by use of the solution from the previous time step, i.e. u^n. In multi-step methods, the computed solutions from *several* previous time steps, e.g., u^n, u^{n-1} and u^{n-2} are used to estimate u^{n+1}. How many time steps that are involved in the computing of u^{n+1}, and how the previous solutions are

combined, depends on the method.[11] With multi-step methods, more than one starting value is required to get the scheme started. Thus, apart from the given initial condition, the remaining starting values must be computed. This is done by some other appropriate scheme (in such a way that the convergence rate of the overall scheme is not reduced).

Note that the Runge-Kutta methods are single-step methods, even if they use several *intermediate* steps (between t_n and t_{n+1}) when computing u^{n+1}, using no other previous solution than u^n.

One of the simplest multi-step methods is the (second order) *two-step Adams-Bashforth method*. The computational scheme reads:

$$u^{n+1} = u^n + \frac{\Delta t}{2}\left(3f(u^n, t_n) - f(u^{n-1}, t_{n-1})\right),$$

for $n = 1, 2, \ldots, N_t - 1$, with $u^0 = U_0$.

a) Implement the scheme in a function `adams_bashforth_2` that takes appropriate parameters, so that it is easy to import and use whenever needed. Use a Forward Euler scheme to compute the missing starting value.

b) Write a function `compare_FE_AdamsBashforth2` that compares graphically the solutions produced by the Forward Euler and two-step Adams-Bashforth methods, when they solve the population growth model $u' = 0.1u$, with $u(0) = 100$. Let the total time span $T = 20$, and use a time step $dt = 2$. In the plot produced, include also the exact solution, so that the numerical solutions can be assessed.

c) Suggest a reasonable asymptotic error model before you write a proper test function `test_convergence_rates` that may be used to compute and check the convergence rates of the implemented AB2 method. However, the test function should take appropriate input parameters, so that it can be used also for other ODE solvers, in particular the `ode_FE` implemented previously (if you already have written this test function when doing Exercise 8.15, you may prefer to import the function).

   Include your test function in a program, together with the two functions you defined previously (`AB2` and `compare_FE_AdamsBashforth2`). Write the code with a test block, so that it gets easy to either import functions from the module, or to run it as a program.

   Finally, run the program (so that `compare_FE_AdamsBashforth2` gets called, as well as `test_convergence_rates` for both FE and AB2) and confirm that it works as expected. In particular, does the plot look good, and do you get the convergence rates you expected for Forward Euler and AB2?

Filename: `Adams_Bashforth_2.py`.

[11] Read more about multi-step methods, e.g., on Wikipedia (https://en.wikipedia.org/wiki/Linear_multistep_method).

Exercise 8.18: The Three-Step Adams-Bashforth Method

This exercise builds on Exercise 8.17, so you better do that one first. Another multi-step method, is the *three-step Adams-Bashforth method*. This is a third order method with a computational scheme that reads:

$$u^{n+1} = u^n + \frac{\Delta t}{12}\Big(23f(u^n, t_n) - 16f(u^{n-1}, t_{n-1}) + 5f(u^{n-2}, t_{n-2})\Big) .$$

for $n = 2, 3, \ldots, N_t - 1$, with $u^0 = U_0$.

a) Assume that someone implemented the scheme as follows:

```
def adams_bashforth_3(f, U_0, dt, T):
    """Third-order Adams-Bashforth scheme for solving first order ODE"""
    N_t = int(round(T/dt))
    u = np.zeros(N_t+1)
    t = np.linspace(0, N_t*dt, len(u))
    u[0] = U_0
    # Compute missing starting values
    u[1] = 100*np.exp(0.1*dt)
    u[2] = 100*np.exp(0.1*(2*dt))
    for n in range(1, N_t, 1):
        u[n+1] = u[n] + (dt/12)*(23*f(u[n], t[n]) - \
                                 16*f(u[n-1], t[n-1]) +\
                                 5*f(u[n-2], t[n-2]))
    return u, t
```

There is one (known!) bug here, find it! Try first by simply reading the code. If not successful, you may try to run it and do some testing on your computer.

Also, what would you say about the way that missing starting values are computed?

b) Repeat Exercise 8.17, using the given three-step method in stead of the two-step method.

Note that with the three-step method, you need 3 starting values. Use the Runge-Kutta third order scheme for this purpose. However, check also the convergence rate of the scheme when missing starting values are computed with Forward Euler in stead.

Filename: `Adams_Bashforth_3.py`.

Exercise 8.19: Use a Backward Euler Scheme for Oscillations

Consider (8.43)–(8.44) modeling an oscillating engineering system. This 2×2 ODE system can be solved by the *Backward Euler scheme*, which is based on discretizing derivatives by collecting information backward in time. More specifically, $u'(t)$ is approximated as

$$u'(t) \approx \frac{u(t) - u(t - \Delta t)}{\Delta t} .$$

A general vector ODE $u' = f(u, t)$, where u and f are vectors, can use this approximation as follows:

$$\frac{u^n - u^{n-1}}{\Delta t} = f(u^n, t_n),$$

which leads to an equation for the new value u^n:

$$u^n - \Delta t f(u^n, t_n) = u^{n-1}\,.$$

For a general f, this is a system of *nonlinear algebraic equations*.

However, the ODE (8.43)–(8.44) is *linear*, so a Backward Euler scheme leads to a system of two algebraic equations for two unknowns:

$$u^n - \Delta t v^n = u^{n-1}, \tag{8.89}$$

$$v^n + \Delta t \omega^2 u^n = v^{n-1}\,. \tag{8.90}$$

a) Solve the system for u^n and v^n.
b) Implement the found formulas for u^n and v^n in a program for computing the entire numerical solution of (8.43)–(8.44).
c) Run the program with a Δt corresponding to 20 time steps per period of the oscillations (see Sect. 8.4.3 for how to find such a Δt). What do you observe? Increase to 2000 time steps per period. How much does this improve the solution?

Filename: `osc_BE.py`.

Remarks While the Forward Euler method applied to oscillation problems $u'' + \omega^2 u = 0$ gives growing amplitudes, the Backward Euler method leads to significantly damped amplitudes.

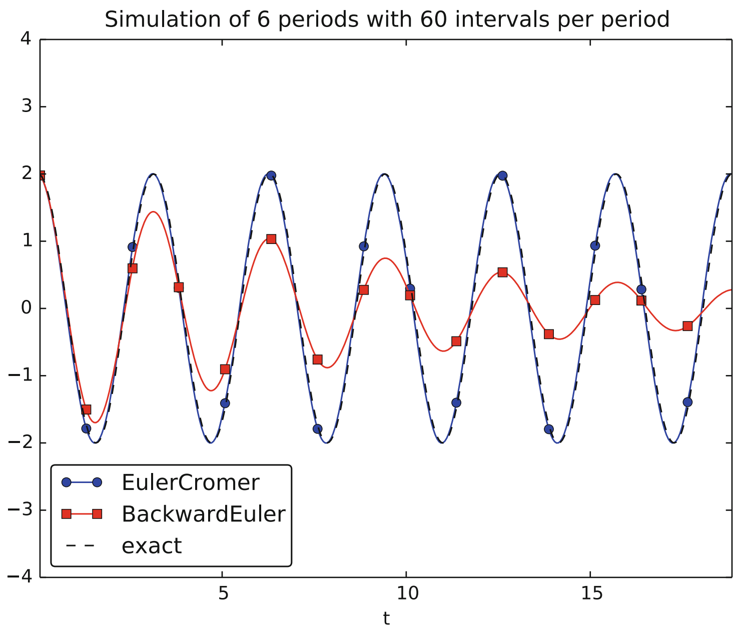
Exercise 8.20: Use Heun's Method for the SIR Model
Make a program that computes the solution of the SIR model from Sect. 8.3.1 both by the Forward Euler method and by Heun's method (or equivalently: the second-order Runge-Kutta method) from Sect. 8.4.5. Compare the two methods in the simulation case from Sect. 8.3.3. Make two comparison plots, one for a large and one for a small time step. Experiment to find what "large" and "small" should be: the large one gives significant differences, while the small one lead to very similar curves.
Filename: `SIR_Heun.py`.

Exercise 8.21: Use Odespy to Solve a Simple ODE
Solve

$$u' = -au + b, \quad u(0) = U_0, \quad t \in (0, T]$$

by the Odespy software. Let the problem parameters a and b be arguments to the function specifying the derivative. Use 100 time intervals in $[0, T]$ and plot the solution when $a = 2$, $b = 1$, $T = 6/a$.
Filename: `odespy_demo.py`.

Exercise 8.22: Set up a Backward Euler Scheme for Oscillations
Write the ODE $u'' + \omega^2 u = 0$ as a system of two first-order ODEs and discretize these with backward differences as illustrated in Fig. 8.22. The resulting method is referred to as a Backward Euler scheme. Identify the matrix and right-hand side of the linear system that has to be solved at each time level. Implement the method, either from scratch yourself or using Odespy (the name is `odespy.BackwardEuler`). Demonstrate that contrary to a Forward Euler scheme, the Backward Euler scheme leads to significant non-physical damping. The figure below shows that even with 60 time steps per period, the results after a few periods are useless:
Filename: `osc_BE.py`.



Exercise 8.23: Set up a Forward Euler Scheme for Nonlinear and Damped Oscillations
Derive a Forward Euler method for the ODE system (8.68)–(8.69). Compare the method with the Euler-Cromer scheme for the sliding friction problem from Sect. 8.4.11:

1. Does the Forward Euler scheme give growing amplitudes?
2. Is the period of oscillation accurate?
3. What is the required time step size for the two methods to have visually coinciding curves?

Filename: `osc_FE_general.py`.

Exercise 8.24: Solving a Nonlinear ODE with Backward Euler

Let y be a scalar function of time t and consider the nonlinear ODE

$$y' + y = ty^3,\ t \in (0,4), \quad y(0) = \frac{1}{2}\,.$$

a) Assume you want to solve this ODE numerically by the Backward Euler method. Derive the computational scheme and show that (contrary to the Forward Euler scheme) you have to *solve a nonlinear algebraic equation* for each time step when using this scheme.

b) Implement the scheme in a program that also solves the ODE by a Forward Euler method. With Backward Euler, use Newton's method to solve the algebraic equation. As your initial guess, you have one good alternative, which one?

   Let your program plot the two numerical solutions together with the exact solution, which is known (e.g., from Wolfram Alpha) to be

$$y(t) = \frac{\sqrt{2}}{\sqrt{7e^{2t} + 2t + 1}}\,.$$

Filename: `nonlinBE.py`.

Exercise 8.25: Discretize an Initial Condition

Assume that the initial condition on u' is nonzero in the finite difference method from Sect. 8.4.12: $u'(0) = V_0$. Derive the special formula for u^1 in this case.
Filename: `ic_with_V_0.pdf`.

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.