Joakim Sundnes*

Solving Ordinary Differential Equations in Python

Jun 6, 2023

*Simula Research Laboratory.

Preface

This book was based on a set of lecture notes written for the book *A Primer on Scientific Programming with Python* by Hans Petter Langtangen [14], mainly covering topics from Appendix A, C, and E. The original notes have been extended with more material on implicit solvers and automatic time-stepping methods, to provide a more complete and balanced overview of state-of-the-art solvers for ordinary differential equations (ODEs). The main purpose of the notes is to serve as a brief and gentle introduction to solving differential equations in Python, for use in the course *Introduction to programming for scientific applications* (IN1900, 10 ETCS credits) at the University of Oslo. To read these notes one should have a basic knowledge of Python and NumPy, see for instance [16], and it is also useful to have a fundamental understanding of ODEs.

One may ask why this is useful to learn how to write your own ODE solvers in Python, when there are already multiple such solvers available, for instance in the SciPy library. However, no single ODE solver is the best and most efficient tool for all possible ODE problems, and the choice of solver should always be based on the characteristics of the problem. To make such choices, it is extremely useful to know the strengths and weaknesses of the different solvers, and the best way to obtain this knowledge is to program your own collection of ODE solvers. Different ODE solvers are also conveniently grouped into families and hierarchies of solvers, and provide an excellent example of how object-oriented programming (OOP) can be used to maximize code reuse and minimize duplication.

The presentation style of the book is compact and pragmatic, and includes a large number of code examples to illustrate how the various ODE solvers can be implemented and applied in practice. The complete source code for all examples, as well as Jupyter notebooks for all the chapters, is provided in the online resources accompanying this book. All the programs and code examples are written in a simple and compact Python style, and generally avoid the use of advanced tools and features. Experienced Python programmers will therefore surely find more elegant and modern solutions to many of

the examples, including, for instance, abstract base classes, type hints, and data classes, to mention a few. However, the main goal of the book is to introduce the fundamentals of ODE solvers and OOP as part of an introductory programming course, and we believe this purpose is best served by focusing on the basics. Readers familiar with scientific computing or numerical software will probably also miss a discussion of computational performance. While performance is clearly relevant when solving ODEs, optimizing the performance of a Python based solver easily becomes quite technical, and requires features like just-in-time compilers (e.g., Numba) or mixed-language programming. The solvers in this book only use fairly basic features of Python and NumPy, which sacrifices some performance but enhances understanding of the solver properties and their implementation.[1]

The book is organized as follows. Chapter 1 introduces the forward Euler method, and uses this simple method to introduce the fundamental ideas and principles that underpin all the methods considered later. The chapter introduces the notation and general mathematical formulation used throughout the book, both for scalar ODEs and systems of ODEs, and is essential reading for everyone with little prior experience with ODEs and ODE solvers. The chapter also briefly explains how to use the ODE solvers from the SciPy library. Readers already familiar with the fundamentals of the forward Euler method and its implementation may consider moving straight to Chapter 2, which presents explicit Runge-Kutta methods. The fundamental ideas of the methods are introduced, and the main focus of the chapter is how a collection of ODE solvers can be implemented as a class hierarchy with minimal code duplication. Chapter 3 introduces so-called *stiff* ODEs, and presents techniques for simple stability analysis of Runge-Kutta methods. The bulk of the chapter is then devoted to programming of implicit Runge-Kutta methods, which have better stability properties than explicit methods and therefore perform better for stiff ODEs. Chapter 4 then concludes the presentation of ODE solvers by introducing methods for adaptive time step control, which is an essential component of all modern ODE software. Chapter 5 is quite different from the preceding ones, since the focus is on a specific class of ODE models rather than a set of solvers. The simpler ODE problems considered in earlier chapters are useful for introducing and testing the solvers, but in order to appreciate both the potential and the challenges of modelling with ODEs it is useful to step beyond this. As an example of a real-world application of ODEs we have chosen the famous Kermack-McKendrick SIR (Susceptible-Infected-Recovered) model from epidemiology. These classic models were first developed in the early 1900s (see [12]), and have received quite some attention in recent years, for obvious reasons. We derive the models from a set of fundamental assumptions, and discuss the implications and limitations resulting from these assumptions. The main focus of the chapter is then on how the models can be modified and extended to capture new phenomena,

[1] Complete source code for all the solvers and examples can be found here: https://sundnes.github.io/solving_odes_in_python/

and how these changes can be implemented and explored using the solvers developed in preceding chapters.

Although the focus of the text is on differential equations, Appendix A is devoted to the related topic of *difference equations*. The motivation for including this chapter is that difference equations are closely related to ODEs, they have many important applications on their own, and numerical methods for ODEs are essentially methods for turning differential equations into difference equations. Solving difference equations can therefore be seen as a natural step on the way towards solving ODEs, and the standard formulation of difference equations in mathematical textbooks is already in a "computer-friendly" form, which is very easy to translate into a Python program using for-loops and arrays. Some students find difference equations easier to understand than differential equations, and may benefit from reading Appendix A first, while others find it easier to go straight to the ODEs and leave Appendix A for later.

April 2023 *Joakim Sundnes*

Contents

Chapter 1
Programming a simple ODE solver

Ordinary differential equations (ODEs) are widely used in science and engineering, in particular for modeling dynamic processes. While simple ODEs can be solved with analytical methods, non-linear ODEs are generally not possible to solve in this way, and we need to apply numerical methods. In this chapter we will see how we can program general numerical solvers that can be applied to any ODE. We will first consider scalar ODEs, i.e., ODEs with a single equation and a single unknown, and in Section 1.3 we will extend the ideas to systems of coupled ODEs. Understanding the concepts of this chapter is useful not only for programming your own ODE solvers, but also for using a wide variety of general-purpose ODE solvers available both in Python and other programming languages.

1.1 Creating a general-purpose ODE solver

When solving ODEs analytically one will typically consider a specific ODE or a class of ODEs, and try to derive a formula for the solution. In this chapter we want to implement numerical solvers that can be applied to any ODE, not restricted to a single example or a particular class of equations. For this purpose, we need a general abstract notation for an arbitrary ODE. We will write the ODEs on the following form:

$$u'(t) = f(t, u(t)), \tag{1.1}$$

which means that the ODE is fully specified by the definition of the right-hand side function $f(t, u)$. Examples of this function may be:

$$f(t,u) = \alpha u, \quad \text{exponential growth}$$
$$f(t,u) = \alpha u\left(1 - \frac{u}{R}\right), \quad \text{logistic growth}$$
$$f(t,u) = -b|u|u + g, \quad \text{falling body in a fluid}$$

Notice that, for generality, we write all these right-hand sides as functions of both t and u, although the mathematical formulations only involve u. This general formulation is not strictly needed in the mathematical equations, but it is very convenient when we start programming, and want to use the same solver for a wide range of ODE models. We will discuss this in more detail later. Our aim is now to write functions and classes that take f as input, and solve the corresponding ODE to produce u as output.

In order for (1.1) to have a unique solution we need to specify the *initial condition* for u, which is the value of the solution at time $t = t_0$. The resulting mathematical problem is written as

$$u'(t) = f(t,u(t)),$$
$$u(t_0) = u_0,$$

and is commonly referred to as an *initial value problem*, or simply an IVP. All the ODE problems we will consider in this book are initial value problems. As an example, consider the very simple ODE

$$u' = u.$$

This equation has the general solution $u(t) = Ce^t$ for any constant C, so it has an infinite number of solutions. Specifying an initial condition $u(t_0) = u_0$ gives $C = u_0$, and we get the unique solution $u(t) = u_0 e^t$. We shall see that, when solving the equation numerically, we need to define u_0 in order to start our method and compute a solution at all.

A simple and general solver; the Forward Euler method. A numerical method for (1.1) can be derived by simply approximating the derivative in the equation $u' = f(t,u)$ by a finite difference. To introduce the idea, assume that we have already computed u at discrete time points $t_0, t_1, \ldots, t_n$. At time t_n we have the ODE

$$u'(t_n) = f(t_n, u(t_n)),$$

and we can now approximate $u'(t_n)$ by a forward finite difference;

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

Inserting this approximation into the ODE at $t = t_n$ yields the following equation

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

and we can rearrange the terms to obtain an explicit formula for $u(t_{n+1})$:

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

This method is known as the *Forward Euler (FE) method* or the *Explicit Euler Method*, and is the simplest numerical method for solving an ODE. The classification as a *forward* or *explicit* method refer to the fact that we have an explicit update formula for $u(t_{n+1})$, which only involves known quantities at time t_n. In contrast, for an *implicit* ODE solver the update formula will include terms on the form $f(t_{n+1}, u(t_{n+1}))$, and we need to solve a generally nonlinear equation to determine the unknown $u(t_{n+1})$. We will visit other explicit ODE solvers in Chapter 2 and implicit solvers in Chapter 3.

To simplify the formula a bit we introduce the notation $u_n = u(t_n)$, i.e., we let u_n denote the numerical approximation to the exact solution $u(t)$ at $t = t_n$. The update formula now reads

$$u_{n+1} = u_n + \Delta t f(t_n, u_n), \tag{1.2}$$

which, if we know the u_0 at time t_0, can be applied repeatedly to u_1, u_2, u_3 and so forth. If we again consider the very simple ODE given by $u' = u$ (i.e., $f(t,u) = u$) we have

$$\begin{aligned} u_1 &= u_0 + \Delta t u_0, \\ u_2 &= u_1 + \Delta t u_1, \\ u_3 &= u_2 + \ldots, \end{aligned}$$

and the general update formula

$$u_{n+1} = u_n + \Delta t u_n = (1 + \Delta t) u_n.$$

In a Python program, the repeated application of the same formula is conveniently implemented in a for-loop, and the solution can be stored in a list or a NumPy array. See, for instance, [16] for an introduction to NumPy arrays and tools, which will be used extensively through these notes. For a given final time T and number of time steps N, we perform the following steps:

1. Create arrays `t` and `u` of length $N+1$[1]
2. Set initial condition: `u[0]` $= u_0$, `t[0]` $= 0$
3. Compute the time step Δt `dt = T/N`
4. For $n = 0, 1, 2, \ldots, N-1$:
   - `t[n + 1] = t[n] + dt`
   - `u[n + 1] = (1 + dt) * u[n]`

[1] For N time steps, the length of the arrays needs to be $N+1$ since we need to store both end points, i.e., $t_0, t_1, \ldots, t_n$ and $u_0, u_1, \ldots, u_n$.

A complete Python implementation of this algorithm may look like

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

N = 20
T = 4
dt = T/N
u0 = 1

t = np.zeros(N + 1)
u = np.zeros(N + 1)

u[0] = u0
for n in range(N):
    t[n + 1] = t[n] + dt
    u[n + 1] = (1 + dt) * u[n]

plt.plot(t, u)
plt.show()
```

Notice that there is no need to set `t[0]= 0` when `t` is created in this way, but updating `u[0]` is important. In fact, forgetting to do so is a very common error in ODE programming, so it is worth taking note of the line `u[0] = u0`. The solution is shown in Figure 1.1, for two different choices of the time step Δt. We see that the approximate solution improves as Δt is reduced, although both the solutions are quite inaccurate. However, reducing the time step further will easily create a solution that cannot be distinguished from the exact solution.

The for-loop in the example above could also be implemented differently, for instance

```
for n in range(1, N+1):
    t[n] = t[n - 1] + dt
    u[n] = (1 + dt) * u[n - 1]
```

Here `n` runs from 1 to `N`, and all the indices inside the loop have been decreased by one so that the end result is the same. In this simple case it is easy to verify that the two loops give the same result, but mixing up the two formulations will easily lead to a loop that runs out of bounds (an `IndexError`), or a loop where the last elements of `t` and `u` are never computed. While seemingly trivial, such errors are very common when programming with for-loops, and it is a good habit to always examine the loop formulation carefully.

Extending the solver to the general ODE. As stated above, the purpose of this chapter is to create general-purpose ODE solvers, that can solve any ODE written on the form $u' = f(t, u)$. This requires a very small modification of the algorithm above;

1. Create arrays `t` and `u` of length $N + 1$

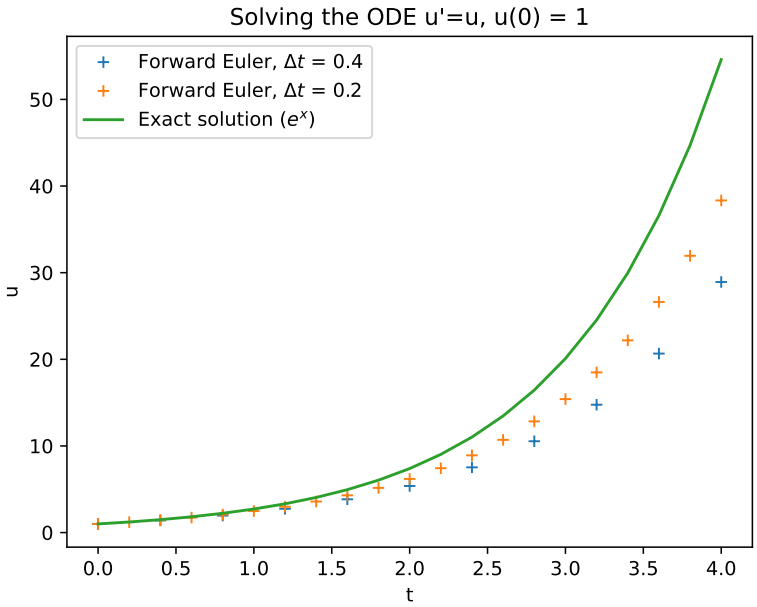

Fig. 1.1 Solution of $u' = u, u(0) = 1$ with $\Delta t = 0.4$ ($N = 10$) and $\Delta t = 0.2$ ($N = 20$).

2. Set initial condition: `u[0]` $= u_0$, `t[0]=0`
3. For $n = 0, 1, 2, \ldots, N-1$:

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

The only change of the algorithm is in the formula for computing `u[n+1]` from `u[n]`. In the previous case we had $f(t,u) = u$, and to create a general-purpose ODE solver we simply replace `u[n]` with the more general `f(t[n],u[n])`. The following Python function implements this generic version of the FE method:

```
import numpy as np

def forward_euler(f, u0, T, N):
    """Solve u'=f(t, u), u(0)=u0, with n steps until t=T."""
    t = np.zeros(N + 1)
    u = np.zeros(N + 1)  # u[n] is the solution at time t[n]

    u[0] = u0
    dt = T / N

    for n in range(N):
        t[n + 1] = t[n] + dt
        u[n + 1] = u[n] + dt * f(t[n], u[n])
```

```
    return t, u
```

This simple function can solve any ODE written on the form (1.1). The right-hand side function $f(t,u)$ needs to be implemented as a Python function, which is then passed as an argument to `forward_euler` together with the initial condition `u0`, the stop time `T` and the number of time steps `N`. The two latter arguments are then used to calculate the time step `dt` inside the function.[2]

To illustrate how the function is used, let us apply it to solve the same problem as above; $u' = u$, $u(0) = 1$, for $t \in [0,4]$. The following code uses the `forward_euler` function to solve this problem:

```
def f(t, u):
    return u

u0 = 1
T = 4
N = 30
t, u = forward_euler(f, u0, T, N)
```

The `forward_euler` function returns the two arrays `u` and `t`, which we can plot or process further as we want. One thing worth noticing in this code is the definition of the right-hand side function `f`. As we mentioned above, this function should always be written with two arguments `t` and `u`, although in this case only `u` is used inside the function. The two arguments are needed because we want our solver to work for all ODEs on the form $u' = f(t,u)$, and the function is therefore called as `f(t[n], u[n])` inside the `forward_euler` function. If our right hand side function was defined as a function of `u` only, i.e., using `def f(u):`, we would get an error message when the function was called inside `forward_euler`. This problem is solved by simply writing `def f(t,u):` even if `t` is never used inside the function.[3]

For being only 15 lines of Python code, the capabilities of the `forward_euler` function above are quite remarkable. Using this function, we can solve any kind of linear or nonlinear ODE, most of which would be impossible to solve using analytical techniques. The general recipe for using this function can be summarized as follows:

1. Identify $f(t,u)$ in your ODE
2. Make sure you have an initial condition u_0
3. Implement the $f(t,u)$ formula in a Python function `f(t, u)`

[2] The source code for this function, as well as all subsequent solvers and examples, can be found here: https://sundnes.github.io/solving_odes_in_python/

[3] This way of defining the right-hand side is a standard used by most available ODE solver libraries, both in Python and other languages. The right-hand side function always takes two arguments `t` and `u`, but, annoyingly, the order of the two arguments varies between different solver libraries. Some expect the `t` argument first, while others expect `u` first.

4. Choose the number of time steps `N`
5. Call `t, u = forward_euler(f, u0, T, N)`
6. Plot the solution

It is worth mentioning that the FE method is the simplest of all ODE solvers, and many will argue that it is not very good. This is partly true, since there are many other methods that are more accurate and more stable when applied to challenging ODEs. We shall see later that numerical solutions may not only be inaccurate, as illustrated in Figure 1.1, but may also blow up and give completely meaningless results. Solvers for avoiding such behavior, i.e., stable solvers, will be presented in Chapter 3. However, the FE method is quite suitable for solving a large number of interesting ODEs. If we are not happy with the accuracy we can simply reduce the time step, and in most cases this will give the accuracy we need with a negligible increase in computing time.

1.2 The ODE solver implemented as a class

We can increase the flexibility of the `forward_euler` solver function by implementing it as a class. There are many ways to implement such a class, but one possible usage can be as follows:

```
method = ForwardEuler_v0(f)
method.set_initial_condition(u0)
t, u = method.solve(t_span=(0, 10), N=100)
plot(t, u)
```

The benefits of using a class instead of a function may not be obvious at this point, but it will become clear when we introduce different ODE solvers later. For now, let us just look at how such a solver class would need to be implemented in order to support the use case specified above:

- We need a constructor (`__init__`) which takes a single argument, the right-hand side function `f`, and stores it as an attribute.
- The method `set_initial_condition` takes the initial condition as argument and stores it.
- The class needs a `solve`-method, which takes the time interval `t_span` and number of time steps `N` as arguments. The method implements the for-loop for solving the ODE and returns the solution, similar to the `forward_euler` function considered earlier.
- The time step Δt and the sequences t_n, u_n must be initialized in one of the methods, and it may also be convenient to store these as attributes. Since the time interval and the number of steps are arguments to the `solve` method it is natural to do these computations there.

In addition to these methods, it may be convenient to implement the formula for advancing the solution one step as a separate method `advance`. In this

way it becomes very easy to implement new numerical methods, since we typically only need to change the `advance` method. A first version of the solver class may look as follows:

```
import numpy as np
class ForwardEuler_v0:
    def __init__(self, f):
        self.f = f

    def set_initial_condition(self, u0):
        self.u0 = u0

    def solve(self, t_span, N):
        """Compute solution for t_span[0] <= t <= t_span[1],
        using N steps."""
        t0, T = t_span
        self.dt = T / N
        self.t = np.zeros(N + 1)  # N steps ~ N+1 time points
        self.u = np.zeros(N + 1)

        msg = "Please set initial condition before calling solve"
        assert hasattr(self, "u0"), msg

        self.t[0] = t0
        self.u[0] = self.u0

        for n in range(N):
            self.n = n
            self.t[n + 1] = self.t[n] + self.dt
            self.u[n + 1] = self.advance()
        return self.t, self.u

    def advance(self):
        """Advance the solution one time step."""
        # Create local variables to get rid of "self." in
        # the numerical formula
        u, dt, f, n, t = self.u, self.dt, self.f, self.n, self.t

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

This class does essentially the same tasks as the `forward_euler` function above, and the main advantage of the class implementation is the increased flexibility that comes with the `advance` method. As we shall see later, implementing a different numerical method typically only requires implementing a new version of this method, while all other code can be left unchanged. Notice also that we have added an `assert` statement inside the `solve` method, which checks that the user has called `set_initial_condition` before calling `solve`. Forgetting to do so is a very likely error for users of the code, and this `assert` statement ensures that we get a useful error message rather than a less informative `AttributeError`.

We can also use a class to hold the right-hand side $f(t, u)$, which is particularly convenient for functions with parameters. Consider for instance the

model for logistic growth;

$$u'(t) = \alpha u(t)\left(1 - \frac{u(t)}{R}\right), \quad u(0) = u_0, \quad t \in [0, 40],$$

which is typically used to model self-limiting growth of a biological population, i.e., growth that is constrained by limited resources. The initial growth is approximately exponential, with growth rate α, and the population curve flattens out as the population size approaches the *carrying capacity* R, see Figure 1.2 for an example solution. The right hand side function includes two parameters α and R, but if we want to solve it using our FE function or class, it must be implemented as a function of t and u only. There are several ways to do this in Python, but one convenient approach is to implement the function as a class with a call method.[4] We can then define the parameters as attributes in the constructor and use them inside the `__call__` method:

```
class Logistic:
    def __init__(self, alpha, R):
        self.alpha, self.R = alpha, float(R)

    def __call__(self, t, u):
        return self.alpha * u * (1 - u / self.R)
```

The main program for solving the logistic growth problem may now look like:

```
problem = Logistic(alpha=0.2, R=1.0)
solver = ForwardEuler_v0(problem)
u0 = 0.1
solver.set_initial_condition(u0)

T = 40
t, u = solver.solve(t_span=(0, T), N=400)
```

1.3 Systems of ODEs

So far we have only considered ODEs with a single solution component, often called scalar ODEs. Many interesting processes can be described by systems of ODEs, i.e., multiple ODEs where the right-hand side of one equation depends on the solution of the others. Such equation systems are also referred to as vector ODEs. One simple example is

$$\begin{aligned} u' &= v, & u(0) &= 1 \\ v' &= -u, & v(0) &= 0. \end{aligned}$$

[4] Recall that if we equip a class with a special method named `__call__`, instances of the class will be callable and behave like regular Python functions. See, for instance, Chapter 8 of [16] for a brief introduction to `__call__` and other special methods.

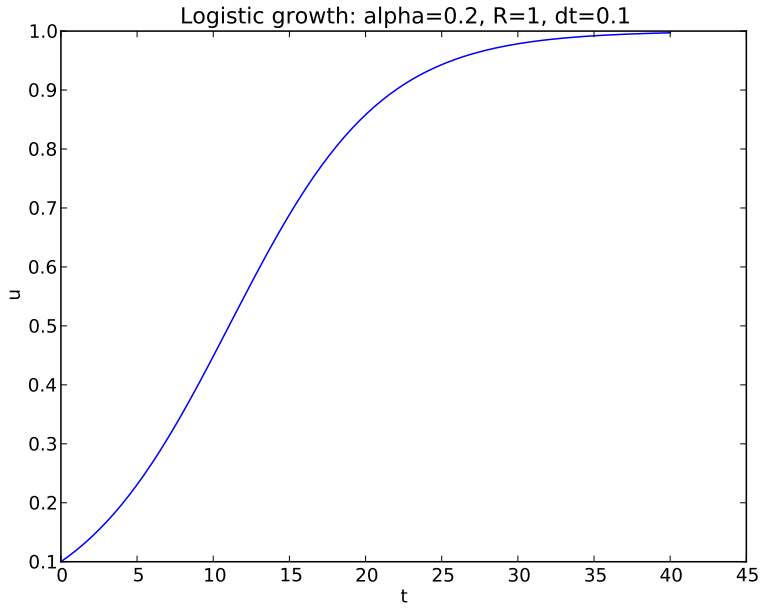

Fig. 1.2 Solution of the logistic growth model.

The solution of this system is $u = \cos t, v = \sin t$, which can easily be verified by inserting the solution into the equations and the initial conditions. For more general cases, it is usually even more difficult to find analytical solutions of ODE systems than of scalar ODEs, and numerical methods are usually required. In this section we will extend the solvers introduced in sections 1.1-1.2 to be able to solve systems of ODEs. We shall see that such an extension requires relatively small modifications of the code.

We want to develop general software that can be applied to any vector ODE or scalar ODE, and for this purpose it is useful to introduce some general mathematical notation. We have m unknowns

$$u^{(0)}(t), u^{(1)}(t), \ldots, u^{(m-1)}(t)$$

in a system of m ODEs:

$$\begin{aligned}
\frac{d}{dt}u^{(0)} &= f^{(0)}(t, u^{(0)}, u^{(1)}, \ldots, u^{(m-1)}),\\
\frac{d}{dt}u^{(1)} &= f^{(1)}(t, u^{(0)}, u^{(1)}, \ldots, u^{(m-1)}),\\
&\vdots = \vdots\\
\frac{d}{dt}u^{(m-1)} &= f^{(m-1)}(t, u^{(0)}, u^{(1)}, \ldots, u^{(m-1)}).
\end{aligned}$$

To simplify the notation (and later the implementation), we collect both the solutions $u^{(i)}(t)$ and right-hand side functions $f^{(i)}$ into vectors;

$$u = (u^{(0)}, u^{(1)}, \ldots, u^{(m-1)}),$$

and

$$f = (f^{(0)}, f^{(1)}, \ldots, f^{(m-1)}).$$

Note that f is now a vector-valued function. It takes $m+1$ input arguments (t and the m components of u) and returns a vector of m values. Using this notation, the ODE system can be written

$$u' = f(t, u), \quad u(t_0) = u_0,$$

where u and f are now vectors and u_0 is a vector of initial conditions. We see that we use exactly the same notation as for scalar ODEs, and whether we solve a scalar or system of ODEs is determined by how we define f and the initial condition u_0. This general notation is completely standard in text books on ODEs, and we can easily make the Python implementation just as general. The generalization of our ODE solvers is facilitated considerably by the convenience of NumPy arrays and vectorized computations.

1.4 A `ForwardEuler` class for systems of ODEs

The `ForwardEuler_v0` class above was written for scalar ODEs, and we now want to make it work for a system $u' = f$, $u(0) = u_0$, where u, f and u_0 are vectors (arrays). To identify how the code needs to be changed, let us first revisit the underlying numerical method. Using the general notation introduced above, applying the forward Euler method to a system of ODEs yields an update formula that looks exactly as for the scalar case, but where all the terms are vectors:

$$\underbrace{u_{k+1}}_{\text{vector}} = \underbrace{u_k}_{\text{vector}} + \Delta t \underbrace{f(u_k, t_k)}_{\text{vector}}.$$

We could also write this formula in terms of the individual components, as in

$$u_{k+1}^{(i)} = u_k^{(i)} + \Delta t f^{(i)}(t_k, u_k), \text{ for } i = 0, \ldots, m-1,$$

but the compact vector notation is much easier to read. Fortunately, the way we write the vector version of the formula is also how NumPy arrays are used in calculations. The Python code for the formula above may therefore look identical to the version for scalar ODEs;

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

with the important difference that both `u[k]`, `u[k+1]`, and `f(t[k], u[k])` are now arrays.[5] Since these are arrays, the solution `u` must be a two-dimensional array, and `u[k],u[k+1]`, etc. are the rows of this array. The function `f` expects an array as its second argument, and must return a one-dimensional array, containing all the right-hand sides $f^{(0)}, \ldots, f^{(n-1)}$. To get a better feel for how these arrays look and how they are used, we may compare the array holding the solution of a scalar ODE to that of a system of two ODEs. For the scalar equation, both `t` and `u` are one-dimensional NumPy arrays, and indexing into `u` gives us numbers, representing the solution at each time step. For instance, in an interactive Python session we may have arrays `t` and `u` with the following contents:

```
>>> t
array([0. ,  0.4, 0.8, 1.2, ... ])
>>> u
array([1. , 1.4, 1.96, 2.744, ... ])
```

and indexing into `u` then gives

```
>>> u[0]
1.0
>>> u[1]
1.4
```

In the case of a system of two ODEs, `t` is still a one-dimensional array, but the solution array `u` is now two-dimensional, with one column for each solution component. We can index it exactly as shown above, and the result is a one-dimensional array of length two, which holds the two solution components at a single time step:

```
>>> u
array([[1.0, 0.8],
       [1.4, 1.1],
       [1.9, 2.7],
       ... ])

>>> u[0]
```

[5] This compact notation requires that the solution vector `u` is represented by a NumPy array. We could, in principle, use lists to hold the solution components, but the resulting code would need to loop over the components and would be far less elegant and readable.

```
array([1.0, 0.8])
>>> u[1]
array([1.4, 1.1])
```

Equivalently, we could write

```
>>> u[0,:]
array([1.0, 0.8])
>>> u[1,:]
array([1.4, 1.1])
```

to make explicit which of the two array dimensions (or *axes*) that we are indexing into.

The similarity of the generic mathematical notation for vector and scalar ODEs, and the convenient algebra of NumPy arrays, indicate that the solver implementation for scalar and system ODEs can also be very similar. This is indeed true, and the `ForwardEuler_v0` class from the previous chapter can be made to work for ODE systems by a few minor modifications:

- Ensure that `f(t,u)` always returns an array.
- Inspect the initial condition `u0` to see if it is a single number (scalar) or a list/array/tuple, and make the array `u` either a one-dimensional or two-dimensional array.[6]

If these two items are handled and initialized correctly, the rest of the code from Section 1.2 will in fact work with no modifications.

The extended class implementation may look like:

```
import numpy as np

class ForwardEuler:
    def __init__(self, f):
        self.f = lambda t, u: np.asarray(f(t, u), float)

    def set_initial_condition(self, u0):
        if np.isscalar(u0):              # scalar ODE
            self.neq = 1                 # no of equations
            u0 = float(u0)
        else:                            # system of ODEs
            self.neq = u0.size           # no of equations
            u0 = np.asarray(u0)
        self.u0 = u0

    def solve(self, t_span, N):
        """Compute solution for
        t_span[0] <= t <= t_span[1],
        using N steps."""
        t0, T = t_span
        self.dt = (T - t0) / N
```

[6] This step is not strictly needed, since we could use a two-dimensional array with shape `(N + 1, 1)` for scalar ODEs. However, using a one-dimensional array for scalar ODEs gives simpler and more intuitive indexing.

```
        self.t = np.zeros(N + 1)
        if self.neq == 1:
            self.u = np.zeros(N + 1)
        else:
            self.u = np.zeros((N + 1, self.neq))

        msg = "Please set initial condition before calling solve"
        assert hasattr(self, "u0"), msg

        self.t[0] = t0
        self.u[0] = self.u0

        for n in range(N):
            self.n = n
            self.t[n + 1] = self.t[n] + self.dt
            self.u[n + 1] = self.advance()
        return self.t, self.u

    def advance(self):
        """Advance the solution one time step."""
        u, dt, f, n, t = self.u, self.dt, self.f, self.n, self.t
        return  u[n] + dt * f(t[n], u[n])
```

It is worth commenting on some parts of this code. First, the constructor looks almost identical to the scalar case, but we use a lambda function and the convenient `np.asarray` function to convert any `f` that returns a list or tuple to a function returning a NumPy array. If `f` already returns an array, `np.asarray` will simply return this array with no changes. This modification is not strictly needed, since we could just assume that the user implements `f` to return an array, but it makes the class more robust and flexible. We have also used the function `isscalar` from NumPy in the `set_initial_condition` method, to check if `u0` is a single number or a NumPy array, and define the attribute `self.neq` to hold the number of equations. The final modification is found in the method `solve`, where the `self.neq` attribute is inspected and `u` is initialized to a one- or two-dimensional array of the correct size. The actual for-loop and the `advance` method are both identical to the previous version of the class.

Example: ODE model for a pendulum. To demonstrate the use of the updated `ForwardEuler` class, we consider a system of ODEs describing the motion of a simple pendulum, as illustrated in Figure 1.3. This nonlinear system is a classic physics problem, and despite its simplicity it is not possible to find an exact analytical solution. We will formulate the system in terms of two main variables; the angle θ and the angular velocity ω, see Figure 1.3. For a simple pendulum with no friction, the dynamics of these two variables is governed by

$$\frac{d\theta}{dt} = \omega, \tag{1.3}$$

$$\frac{d\omega}{dt} = -\frac{g}{L}\sin(\theta), \tag{1.4}$$

where L is the length of the pendulum and g is the gravitational constant. Eq. (1.3) follows directly from the definition of the angular velocity, while (1.4) follows from Newton's second law, where $d\omega/dt$ is the acceleration and the right-hand side is the tangential component of the gravitational force acting on the pendulum, divided by its mass. To solve the system we need to define initial conditions for both unknowns, i.e., we need to know the initial position and velocity of the pendulum.

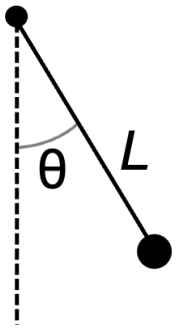

Fig. 1.3 Illustration of the pendulum problem. The main variables of interest are the angle θ and its derivative ω (the angular velocity).

Since the right-hand side defined by (1.3)-(1.4) includes the parameters L and g, it is convenient to implement it as a class, as illustrated for the logistic growth model earlier. A possible implementation may look like this:

```
from math import sin

class Pendulum:
    def __init__(self, L, g=9.81):
        self.L = L
        self.g = g

    def __call__(self, t, u):
        theta, omega = u
        dtheta = omega
        domega = -self.g / self.L * sin(theta)
        return [dtheta, domega]
```

We see that the function returns a list, but this will automatically be wrapped into a function returning an array by the solver class' constructor, as mentioned above. The main program is not very different from the examples of the previous chapter, except that we need to define an initial condition with two

components. Assuming that this class definition as well as the `ForwardEuler` exist in the same file, the code to solve the pendulum problem can look like this:

```
import matplotlib.pyplot as plt

problem = Pendulum(L=1)
solver = ForwardEuler(problem)
solver.set_initial_condition([np.pi / 4, 0])
T = 10
N = 1000
t, u = solver.solve(t_span=(0, T), N=N)

plt.plot(t, u[:, 0], label=r'$\theta$')
plt.plot(t, u[:, 1], label=r'$\omega$')
plt.xlabel('t')
plt.ylabel(r'Angle ($\theta$) and angular velocity ($\omega$)')
plt.legend()
plt.show()
```

Notice that to extract each solution component we need to index into the second index of `u`, using array slicing. Indexing with the first index, for instance using `u[0]` or `u[0,:]`, would give us an array of length two that contains the solution components at the first time point. In this specific example a call like `plt.plot(t, u)` would also work, and would plot both solution components. However, we are often interested in plotting selected components of the solution, and in this case the array slicing is needed. The resulting plot is shown in Figure 1.4. Another minor detail worth noticing is use of Python's raw string format for the labels, indicated by the `r` in front of the string. Raw strings will treat the backslash (\) as a regular character, and is often needed when using Latex encoding of mathematical symbols. The observant reader may also notice that the amplitude of the pendulum motion appears to increase over time, which is clearly not physically correct. In fact, for an undamped pendulum problem defined by (1.3)-(1.4), the energy is conserved, and the amplitude should therefore be constant. The increasing amplitude is a numerical artefact introduced by the forward Euler method, and the solution may be improved by reducing the time step or replacing the numerical method.

1.5 Checking the error in the numerical solution

Recall from Section 1.1 that we derived the FE method by approximating the derivative with a finite difference;

$$u'(t_n) \approx \frac{u(t_{n+1}) - u(t_n)}{\Delta t}. \tag{1.5}$$

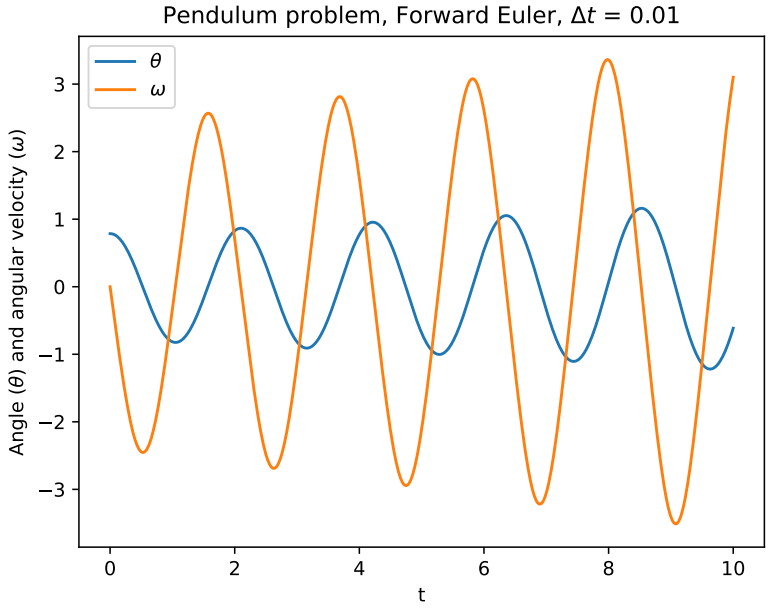

Fig. 1.4 Solution of the simple pendulum problem, computed with the forward Euler method.

This approximation obviously introduces an error, and since we approach the true derivative as $\Delta t \to 0$ it is quite intuitive that the error depends on the size of the time step Δt. This relation was demonstrated visually in Figure 1.1, but it would of course be valuable to have a more precise quantification of how the error depends on the time step. Analyzing the error in numerical methods is a large branch of applied mathematics, which we will not cover in detail here, and the interested reader is referred to, for instance [8]. However, when implementing a numerical method it is very useful to know its theoretical accuracy, and in particular to be able to compute the error and verify that it behaves as expected.

The Taylor expansion, also discussed briefly in Appendix A.4, is an essential tool for estimating the error in numerical methods for ODEs. For any smooth function $\hat{u}(t)$, if we can compute the function value and its derivatives at t_n, the value at $t_n + \Delta t$ can be approximated by the series

$$\hat{u}(t_n + \Delta t) = \hat{u}(t_n) + \Delta t \hat{u}'(t_n) + \frac{\Delta t^2}{2}\hat{u}''(t_n) + \frac{\Delta t^3}{6}\hat{u}'''(t_n) + O(\Delta t^4).$$

We can include as many terms as we like, and since Δt is small we always have $\Delta t^{(n+1)} \ll \Delta t^n$, so the error in the approximation is dominated by the first neglected term. The update formula of the FE method, derived from

(1.5), was

$$u_{n+1} = u(t_n) + \Delta t u'(t_n),$$

which we may recognize as a Taylor series truncated after the first order term, and we expect the error $|u_{n+1} - \hat{u}_{n+1}|$ to be proportional to Δx^2. Since this is the error for a single time step, the accumulated error after $N \sim 1/\Delta t$ steps is proportional to Δt, and the FE method is hence a *first order* method. As we will see in Chapter 2, more accurate methods can be constructed by deriving update formulas that make more terms in the Taylor expansion of the error cancel. This process is fairly straightforward for low-order methods, e.g., of second or third order, but it quickly gets complicated for high order solvers, see, for instance [8] for details.

Knowing the theoretical accuracy of an ODE solver is important for a number of reasons, and one of them is that it provides a method for verifying our implementation of the solver. If we can solve a given problem and demonstrate that the error behaves as predicted by the theory, it gives a good indication that the solver is implemented correctly. We can illustrate this procedure using the simple initial value problem introduced earlier;

$$u' = u, \quad u(0) = 1.$$

As stated above, this problem has the analytical solution $u = e^t$, and we can use this to compute the error in our numerical solution. But how should the error be defined? There is no unique answer to this question. For practical applications, the so-called root-mean-square (RMS) or relative-root-mean-square (RRMS) are commonly used error measures, defined by

$$RMSE = \sqrt{\frac{1}{N}\sum_{n=0}^{N}(u_n - \hat{u}(t_n))^2},$$

$$RRMSE = \sqrt{\frac{1}{N}\sum_{n=0}^{N}\frac{(u_n - \hat{u}(t_n))^2}{\hat{u}(t_n)^2}},$$

respectively. Here, u_n is the numerical solution at time step n and $\hat{u}(t_n)$ the corresponding exact solution. In more mathematically oriented texts, the errors are usually defined in terms of *norms*, for instance the discrete l_1, l_2, and l_∞ norms:

$$e_{l_1} = \sum_{i=0}^{N} (|u_i - \hat{u}(t_i)|),$$
$$e_{l_2} = \sum_{i=0}^{N} (u_i - \hat{u}(t_i)^2),$$
$$e_{l_\infty} = \max_{i=0}^{N} (\hat{u}_i - u(t_i)).$$

While the choice of error norm may be important for certain cases, for practical applications it is usually not very important, and all the different error measures can be expected to behave as predicted by the theory. For simplicity, we will apply an even simpler error measure for our example, where we simply compute the error at the final time T, given by $e = |u_N - \hat{u}(t_N)|$. Using the `ForwardEuler` class introduced above, the complete code for checking the convergence may look as follows:

```
from forward_euler_class_v1 import ForwardEuler
import numpy as np

def rhs(t, u):
    return u

def exact(t):
    return np.exp(t)

solver = ForwardEuler(rhs)
solver.set_initial_condition(1.0)

T = 3.0
t_span = (0,T)
N = 30

print('Time step (dt)   Error (e)      e/dt')
for _ in range(10):
    t, u = solver.solve(t_span, N)
    dt = T / N
    e = abs(u[-1] - exact(T))
    print(f'{dt:<14.7f}   {e:<12.7f}   {e/dt:5.4f}')
    N = N * 2
```

Most of the lines are identical to the previous programs, but we have put the call to the `solve` method inside a for loop, and the last line ensures that the number of time steps `N` is doubled for each iteration of the loop. Notice also the f-string format specifiers, for instance `{dt:<14.7f}`, which sets the output to be a left aligned decimal number with seven decimals, and occupying 14 characters in total. The purpose of these specifiers is to output the numbers as vertically aligned columns, which improves readability and may be important for visually inspecting the convergence. See, for instance, [16] for a brief

introduction to f-strings and format specifiers. The program will produce the following output:

```
Time step (dt)   Error (e)      e/dt
0.1000000        2.6361347      26.3613
0.0500000        1.4063510      28.1270
0.0250000        0.7273871      29.0955
0.0125000        0.3700434      29.6035
0.0062500        0.1866483      29.8637
0.0031250        0.0937359      29.9955
0.0015625        0.0469715      30.0618
0.0007813        0.0235117      30.0950
0.0003906        0.0117624      30.1116
0.0001953        0.0058828      30.1200
```

In the rightmost column we see that the error divided by the time step is approximately constant, which supports the theoretical result that the error is proportional to Δt. In subsequent chapters we will perform similar calculations for higher order methods, to confirm that the error is proportional to Δt^r, where r is the theoretical order of convergence for the method.

In order to compute the error in our numerical solution we obviously need to know the true solution to our initial value problem. This part was easy for the simple example above, since we knew the analytical solution to the equation, but this solution is only available for very simple ODE problems. It is, however, often interesting to estimate the error and the order of convergence for more complex problems, and for this task we need to take a different approach. Several alternatives exist, including for instance the *method of manufactured solutions*, where one simply choses a solution function $u(t)$ and computes its derivative analytically to determine the right-hand side of the ODE. An even simpler approach, which usually works well, is to compute a very accurate numerical solution using a high-order solver and small time steps, and use this solution as the reference for computing the error. For accurate error estimates it is of course essential that the reference solution is considerably more accurate than the numerical solution we want to evaluate. The reference solution therefore typically requires very small time steps and can take some time to compute, but in most cases the computation time will not be a problem.

1.6 Using ODE solvers from SciPy

As mentioned in the preface to this book, there are numerous ODE solvers around that can be used directly, so one may argue that there is no need to implement our own solvers. This may indeed be true, but, as we have argued earlier, it is sometimes very useful to know the inner workings of the solvers we apply, and the best way to obtain this knowledge is to implement the solvers ourselves. However, if we have a given ODE model and want to solve

it as efficiently as possible, there are several existing solvers to choose from. For Python programmers, the most natural choice may be the solvers from *SciPy*, which have evolved into a robust and fairly efficient suite of ODE solvers. SciPy is a large suite of scientific software in Python, including tools for linear algebra, optimization, integration, and other common tasks of scientific computing.[7] For solving initial value problems, the tool of choice is the `solve_ivp` function from the `integrate` module. The following code applies `solve_ivp` with the `Pendulum` class presented above to solve the simple pendulum problem defined by (1.3)-(1.4). We assume that the `Pendulum` class is saved in a separate file `pendulum.py`.

```
from scipy.integrate import solve_ivp
import numpy as np
import matplotlib.pyplot as plt
from pendulum import Pendulum

problem = Pendulum(L = 1)
t_span = (0, 10.0)
u0 = (np.pi/4, 0)

solution = solve_ivp(problem, t_span, u0)

plt.plot(solution.t, solution.y[0,:])
plt.plot(solution.t, solution.y[1,:])
plt.legend([r'$\theta$',r'$\omega$'])
plt.show()
```

Running this code will result in a plot similar to Figure 1.5, and we observe that the solution does not look nearly as nice as the one we got from the `ForwardEuler` solver above. The reason for this apparent error is that `solve_ivp` is an *adaptive* solver, which chooses the time step automatically to satisfy a given error tolerance. The default value of this tolerance is relatively large, which results in the solver using very few time steps and the solution plots looking jagged. If we compare the plot with a very accurate numerical solution, indicated by the two dotted curves in Figure 1.5, we see that the solution at the time points t_n is quite accurate, but the linear interpolation between the time points completely destroys the visual appearance. A more visually appealing solution can be obtained in several ways. We may, for instance, pass the function an additional argument `t_eval`, which is a NumPy array containing the time points where we want to evaluate the solution:

```
t_eval = np.linspace(0, 10.0, 1001)
solution = solve_ivp(problem, t_span, u0,t_eval=t_eval)
```

Alternatively, we can reduce the error tolerance of the solver, for instance setting

```
rtol = 1e-6
solution = solve_ivp(problem, t_span, u0, rtol=rtol)
```

[7] See `https://scipy.org/`

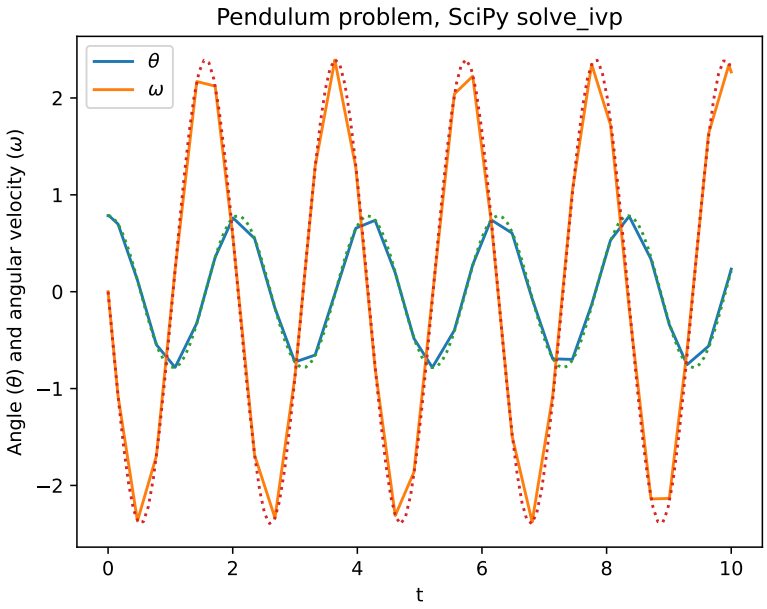

Fig. 1.5 Solution of the simple pendulum problem, computed with the SciPy `solve_ivp` function and the default tolerance.

This latter call will reduce the relative tolerance `rtol` from its default value of `1e-3` (0.001). We could also adjust the absolute tolerance using the parameter `atol`. While we will not consider all the possible arguments and options to `solve_ivp` here, we mention that we can also change the numerical method used by the function, by passing in a parameter named `method`. For instance, a call like

```
rtol = 1e-6
solution = solve_ivp(problem, t_span, u0, method='Radau')
```

will replace the default solver (called `rk45`) with an implicit Radau ODE solver, which we will introduce and explain in Chapter 3. For a complete description of parameters accepted by the `solve_ivp` function we refer to the online SciPy documentation.

Chapter 2
Improving the accuracy

As mentioned earlier, the FE method derived in Chapter 1 is not the most sophisticated ODE solver, although it is sufficiently accurate for most of the applications we will consider in this book. Many alternative methods exist, which have better accuracy and stability and are therefore better suited for solving challenging ODE systems. In this chapter we will focus on improving the accuracy, which means that we will stick with explicit methods. Implicit methods, which have better stability properties and are more suitable for so-called *stiff* ODEs, will be considered in Chapter 3.

In Chapter 1 we demonstrated that the FE method is a first order accurate method, which means that the error in the numerical solution is proportional to the time step size Δt. In this chapter we will derive solvers of higher order, for which the numerical error is proportional to a higher power of Δt. To illustrate how such higher order ODE solvers can be derived, we return to the general formulation of the ODE system:

$$u' = f(t, u), \quad u(t_0) = u_0.$$

We derived the FE method by simply replacing the derivative with a finite difference approximation, but in the present chapter we will apply a slightly different approach to motivate higher order solvers. Assuming that we know the solution u_n at time t_n, the solution at time t_{n+1} can be found by integrating both sides of the equation. We have

$$\int_{t_n}^{t_{n+1}} \frac{du}{dt} dt = \int_{t_n}^{t_{n+1}} f(t, u(t)) dt,$$

which gives us the exact solution at time t_{n+1} as

$$u(t_{n+1}) = u(t_n) + \int_{t_n}^{t_{n+1}} f(t, u(t)) dt. \tag{2.1}$$

It is in general not possible to compute the integral on the right-hand side analytically, since f is often non-linear and the function $u(t)$ is unknown. However, we can approximate the integral using a variety of numerical integration techniques. The simplest approximation is to set

$$f(t, u(t)) \approx f(t_n, u_n), \text{ for } t_n < t < t_{n+1},$$

i.e., approximate the integrand as a constant on the interval t_n to $t_n + \Delta t$. Inserting this choice in (2.1) gives the update formula

$$u_{n+1} = u_n + \Delta t\, f(t_n, u_n),$$

which we recognize as the FE method introduced in Chapter 1. Approximating the function $f(t_n, u_n)$ as constant on the interval $t_n < t < t_{n+1}$ is obviously not the most accurate choice, and we shall see that more accurate approximations of this integrand gives rise to ODE solvers of higher order.

The classical way to approximate the integral of a general non-linear function is to approximate the function by a polynomial, and then integrating this polynomial analytically. This approach forms the basis for standard quadrature rules for numerical integration, and has also been used to derive accurate ODE solvers. Two main ideas have been explored for constructing the polynomial approximation of $f(t, u)$, and have led to two important classes of ODE solvers. The first approach is to approximate $f(t, u)$ by a polynomial which interpolates f in previous time points, i.e., $f(t_{n-1}, u_{n-1}), f(t_{n-2}, u_{n-2}), \ldots$. This method gives rise to the so-called *multistep* methods, which are widely used for solving ODEs. We will not consider multistep methods in this book, but the interested reader is referred to, for instance, [1, 8]. The second approach is to compute a number of intermediate approximations of $f(t, u)$ on the interval $t_n < t < t_{n+1}$, and use these values to define the polynomial approximation of f. This idea is similar to how classical quadrature rules for numerical integration are derived, and gives rise to a class of ODE solvers known as Runge-Kutta methods. Runge-Kutta methods come in many forms, with very different accuracy and stability properties, and will be the main topic of chapters 2-4.

2.1 Explicit Runge-Kutta methods

An intuitive way to improve the accuracy of the approximate integral in (2.1) is to compute a number of intermediate approximations of $f(t_*, u_*)$ for $t_n \leq t_* \leq t_{n+1}$, and compute the integral as a weighted sum of these values. This approach builds on standard techniques of numerical integration, and gives rise to a very popular class of ODE solvers known as *Runge-Kutta methods*. The simplest example of a Runge-Kutta method is in fact the FE

method, which is an example of a one-stage, first-order, explicit Runge-Kutta method. An alternative formulation of the FE method is

$$\begin{aligned} k_1 &= f(t_n, u_n), \\ u_{n+1} &= u_n + \Delta t k_1. \end{aligned}$$

It can easily be verified that this is the same formula as introduced above, and there is no real benefit from writing the formula in two lines rather than one. However, this second formulation is more in line with how Runge-Kutta methods are usually written, and it makes it easy to see the relation between the FE method and more advanced solvers. The intermediate value k_1 is often referred to as a *stage derivative* in the ODE literature.

We can easily improve the accuracy of the FE method to second order, i.e., error proportional to Δt^2, by introducing more accurate approximations of the integral in (2.1). One option is to keep the assumption that $f(t, u(t))$ is constant over $t_n \leq t_* \leq t_{n+1}$, but to approximate it at the middle of the interval rather than the left end. This approach requires one additional stage:

$$k_1 = f(t_n, u_n), \tag{2.2}$$

$$k_2 = f(t_n + \frac{\Delta t}{2}, u_n + \frac{\Delta t}{2} k_1), \tag{2.3}$$

$$u_{n+1} = u_n + \Delta t\, k_2. \tag{2.4}$$

This method is known as the explicit midpoint method or the modified Euler method. The first step is identical to that of the FE method, but instead of using the stage derivate k_1 to advance the solution to the next step, we use it to compute an intermediate midpoint solution

$$u_{n+1/2} = u_n + \frac{\Delta t}{2} k_1.$$

This solution is then used to compute the corresponding stage derivative k_2, which becomes an approximation to the derivative of u at time $t_n + \Delta t/2$. Finally, we use this midpoint derivative to advance the solution to t_{n+1}.

An alternative second order method is Heun's method, also referred to as the explicit trapezoidal method, which can be derived by approximating the integral in (2.1) by a trapezoidal rule:

$$k_1 = f(t_n, u_n), \tag{2.5}$$

$$k_2 = f(t_n + \Delta t, u_n + \Delta t k_1), \tag{2.6}$$

$$u_{n+1} = u_n + \frac{\Delta t}{2}(k_1 + k_2). \tag{2.7}$$

This method also computes two stage derivatives k_1 and k_2, but from the formula for k_2 we see that it approximates the derivative at t_{n+1} rather than

at the midpoint $t_n + \Delta t/2$. The solution is then advanced from t_n to t_{n+1} using the mean value of k_1 and k_2.

All Runge-Kutta methods follow the same recipe as the two second-order methods considered above; we compute one or more intermediate values (i.e., stage derivatives), and then advance the solution using a combination of these stage derivatives. The accuracy of the method can be improved by adding more stages. A general Runge-Kutta method with s stages can be written as

$$k_i = f(t_n + c_i \Delta t, u_n + \Delta t \sum_{j=1}^{s} a_{ij} k_j), \text{ for } i = 1, \ldots, s \tag{2.8}$$

$$u_{n+1} = u_n + \Delta t \sum_{i=1}^{s} b_i k_i. \tag{2.9}$$

Here c_i, b_i, a_{ij}, for $i, j, = 1, \ldots, s$ are method-specific coefficients. All Runge-Kutta methods can be written in this form, and a method is uniquely determined by the number of stages s and the values of the coefficients.

As mentioned above, there exists a wide variety of Runge-Kutta methods, where the coefficients are typically chosen to optimize the accuracy for a given number of stages. We will not dive into the details of how the methods are constructed here, but some of the principles can be quite useful to know about. For instance, it can be shown that the b_i coefficients must be chosen to satisfy $\sum_{i=1}^{s} b_i = 1$ in order to have a convergent method. This condition follows quite naturally from our motivation for Runge-Kutta methods as numerical integrators applied to (2.1). When approximating the integral as a weighted sum, the sum of the weights must obviously be one. Another common constraint on the coefficients is to set $c_i = \sum_{j=1}^{s} a_{ij}$. While this constraint is not strictly needed, it may simplify the derivation of the methods and follows naturally from our interpretation of the stage derivative k_i as approximations of the right-hand side $f(t, u)$ at time $t_n + c_i \Delta t$. When implementing a new solver it is very easy to introduce errors in the coefficient values, and it may be useful to include tests to verify that the most fundamental conditions on the coefficients are satisfied. It is possible to derive general *order conditions* that the coefficients must satisfy for a method to be of a given order, see, for instance, [1, 8] for details. In this chapter we only consider explicit Runge-Kutta methods, which means that $a_{ij} = 0$ for $j \geq i$. It can be shown that the order p of an explicit Runge-Kutta method with s stages satisfies $p \leq s$, and for $p \geq 5$ the bound is $p \leq s - 1$. However, it is not known whether the last bound is sharp, and it may be even stricter for methods of very high order. For instance, all known methods with $p = 8$ have at least eleven stages, and it is not known whether eight-order methods with nine or ten stages exist.

In the ODE literature the method coefficients are often specified in the form of a *Butcher tableau*, which offers a compact definition of any Runge-Kutta method. The Butcher tableau is simply a specification of all the method

coefficients, and for a general Runge-Kutta method it is written as

$$\begin{array}{c|ccc} c_i & a_{11} & \cdots & a_{1s} \\ \vdots & \vdots & & \vdots \\ c_s & a_{s1} & \cdots & a_{ss} \\ \hline & b_1 & \cdots & b_s \end{array}$$

The Butcher tableaus of the three methods considered above; FE, explicit midpoint, and Heun's method, are

$$\begin{array}{c|c} 0 & 0 \\ \hline & 1 \end{array}, \qquad \begin{array}{c|cc} 0 & 0 & 0 \\ 1/2 & 1/2 & 0 \\ \hline & 0 & 1 \end{array}, \qquad \begin{array}{c|cc} 0 & 0 & 0 \\ 1 & 1 & 0 \\ \hline & 1/2 & 1/2 \end{array},$$

respectively. To grasp the concept of Butcher tableaus, it is a good exercise to insert the coefficients from these three tableaus into (2.8)-(2.9) and verify that you arrive at the correct formulae for the three methods. As an example of a higher order method, we may consider the "original" Runge-Kutta method, which is a fourth-order, four-stage method defined by

$$\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ 1/2 & 1/2 & 0 & 0 & 0 \\ 1/2 & 0 & 1/2 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ \hline & 1/6 & 1/3 & 1/3 & 1/6 \end{array},$$

which gives the formulas

$$k_1 = f(t_n, u_n), \tag{2.10}$$

$$k_2 = f(t_n + \frac{\Delta t}{2}, u_n + \frac{\Delta t}{2} k_1), \tag{2.11}$$

$$k_3 = f(t_n + \frac{\Delta t}{2}, u_n + \frac{\Delta t}{2} k_2), \tag{2.12}$$

$$k_4 = f(t_n + \Delta t, u_n + \Delta t k_3), \tag{2.13}$$

$$u_{n+1} = u_n + \frac{\Delta t}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right). \tag{2.14}$$

As mentioned above, all the methods considered in this chapter are explicit methods, which means that $a_{ij} = 0$ for $j \geq i$. As may be observed from (2.10)-(2.14), or from a more careful inspection of the general formula (2.8), this means that the expression for computing each stage derivative k_i only includes previously computed stage derivatives. Therefore, all k_i can be computed sequentially using explicit formulae. For implicit Runge-Kutta methods, on the other hand, we have $a_{ij} \neq 0$ for some $j \geq i$. We see from (2.8) that the formula for computing k_i will then include k_i on the right-hand side, as

part of the argument to the function f. We therefore need to solve equations to compute the stage derivatives, and since f is typically non-linear we need to solve these equations with an iterative solver such as Newton's method. These steps make implicit Runge-Kutta methods more complex to implement and more computationally expensive per time step, but they are also more stable than explicit methods and perform much better for certain classes of ODEs. We will consider implicit Runge-Kutta methods in Chapter 3.

2.2 A class hierarchy of Runge-Kutta methods

We now want to implement Runge-Kutta methods as classes, similar to the FE classes introduced above. When inspecting the `ForwardEuler` class, we quickly observe that most of the code is common to all ODE solvers, and not specific to the FE method. For instance, we always need to create an array for holding the solution, and the general solution method using a for-loop is always the same. In fact, the only difference between the different methods is how the solution is advanced from one step to the next. Recalling the ideas of Object-Oriented Programming, it becomes obvious that a class hierarchy is convenient for implementing such a collection of ODE solvers. In this way we can collect all common code in a superclass (base class), and rely on inheritance to avoid code duplication. The superclass can handle most of the more administrative steps of the ODE solver, such as

- Storing the solution u_n and the time points t_n, $k = 0, 1, 2, \ldots, n$
- Storing the right-hand side function $f(t, u)$
- Storing and applying the initial condition
- Running the loop over all time steps

We can introduce a superclass `ODESolver` to handle these parts, and implement the method-specific details in subclasses. It should now become quite obvious why we chose to isolate the code to perform a single step in the `advance` method, since this will then be the only method we need to implement in the subclasses. The implementation of the superclass can be quite similar to the `ForwardEuler` class introduced earlier:

```
import numpy as np

class ODESolver:
    def __init__(self, f):
        # Wrap user's f in a new function that always
        # converts list/tuple to array (or let array be array)
        self.model = f
        self.f = lambda t, u: np.asarray(f(t, u), float)

    def set_initial_condition(self, u0):
        if np.isscalar(u0):              # scalar ODE
```

```
            self.neq = 1                # no of equations
            u0 = float(u0)
        else:                           # system of ODEs
            u0 = np.asarray(u0)
            self.neq = u0.size          # no of equations
        self.u0 = u0

    def solve(self, t_span, N):
        """Compute solution for t_span[0] <= t <= t_span[1],
        using N steps."""
        t0, T = t_span
        self.dt = (T - t0) / N
        self.t = np.zeros(N + 1)  # N steps ~ N+1 time points
        if self.neq == 1:
            self.u = np.zeros(N + 1)
        else:
            self.u = np.zeros((N + 1, self.neq))

        msg = "Please set initial condition before calling solve"
        assert hasattr(self, "u0"), msg

        self.t[0] = t0
        self.u[0] = self.u0

        for n in range(N):
            self.n = n
            self.t[n + 1] = self.t[n] + self.dt
            self.u[n + 1] = self.advance()
        return self.t, self.u

        def advance(self):
            raise NotImplementedError(
                "Advance method is not implemented in the base class")
```

Notice that the `ODESolver` is meant to be a pure superclass, and the implementation of the `advance` method is left for subclasses. In order to make this abstract nature of the class explicit, we have implemented an `advance` method that will simply raise a `NotImplementedError` when it is called. If we try to make an instance of `ODESolver` and use it as a stand-alone solver, we will get an error in the line `self.u[n + 1] = self.advance()`. If we had left out the definition of `advance` completely we would get an error from the same line, but it would be a less informative `AttributeError`. Raising the `NotImplementedError` makes it clear to anyone reading or using the code that this behavior is intentional, and that the functionality is to be implemented in subclasses. It should be noted that there are alternative ways in Python to make explicit the abstract nature of the `ODESolver` class, for instance using the module `abc`, for "Abstract Base Class". However, while this solution may be considered more modern, we have decided to not use it here, in the interest of keeping the code simple and compact.

It is also worth commenting on the `solve` method,

With the superclass at hand, the implementation of a `ForwardEuler` subclass becomes very simple:

```
class ForwardEuler(ODESolver):
    def advance(self):
        u, f, n, t = self.u, self.f, self.n, self.t
        dt = self.dt
        return u[n] + dt * f(t[n], u[n])
```

Similarly, the explicit midpoint method and the fourth-order Runge-Kutta method can be subclasses, each implementing a single method:

```
class ExplicitMidpoint(ODESolver):
    def advance(self):
        u, f, n, t = self.u, self.f, self.n, self.t
        dt = self.dt
        dt2 = dt / 2.0
        k1 = f(t[n], u[n])
        k2 = f(t[n] + dt2, u[n] + dt2 * k1)
        return u[n] + dt * k2

class RungeKutta4(ODESolver):
    def advance(self):
        u, f, n, t = self.u, self.f, self.n, self.t
        dt = self.dt
        dt2 = dt / 2.0
        k1 = f(t[n], u[n],)
        k2 = f(t[n] + dt2, u[n] + dt2 * k1, )
        k3 = f(t[n] + dt2, u[n] + dt2 * k2, )
        k4 = f(t[n] + dt, u[n] + dt * k3, )
        return u[n] + (dt / 6.0) * (k1 + 2 * k2 + 2 * k3 + k4)
```

The use of these classes is nearly identical to the FE class introduced in Section 1.3. Considering the same simple ODE used above; $u' = u$, $u(0) = 1$, $t \in [0,3]$, $\Delta t = 0.5$, the code looks like:

```
import numpy as np
import matplotlib.pyplot as plt
from ODESolver import ForwardEuler, ExplicitMidpoint, RungeKutta4

def f(t, u):
    return u

t_span = (0, 3)
N = 6

fe = ForwardEuler(f)
fe.set_initial_condition(u0=1)
t1, u1 = fe.solve(t_span, N)
plt.plot(t1, u1, label='Forward Euler')

em = ExplicitMidpoint(f)
em.set_initial_condition(u0=1)
t2, u2 = em.solve(t_span, N)
```

```
plt.plot(t2, u2, label='Explicit Midpoint')

rk4 = RungeKutta4(f)
rk4.set_initial_condition(u0=1)
t3, u3 = rk4.solve(t_span, N)
plt.plot(t3, u3, label='Runge-Kutta 4')

# plot the exact solution in the same plot
time_exact = np.linspace(0, 3, 301)
plt.plot(time_exact, np.exp(time_exact), label='Exact')
plt.title('RK solvers for exponential growth, $\\Delta t = 0.5$')
plt.xlabel('$t$')
plt.ylabel('$u(t)$')
plt.legend()
plt.show()
```

This code will solve the same simple equation using three different methods, and plot the solutions in the same window, as shown in Figure 2.1. We set `N = 6`, which corresponds to a very large time step ($\Delta t = 0.5$), to highlight the difference in accuracy between the methods.

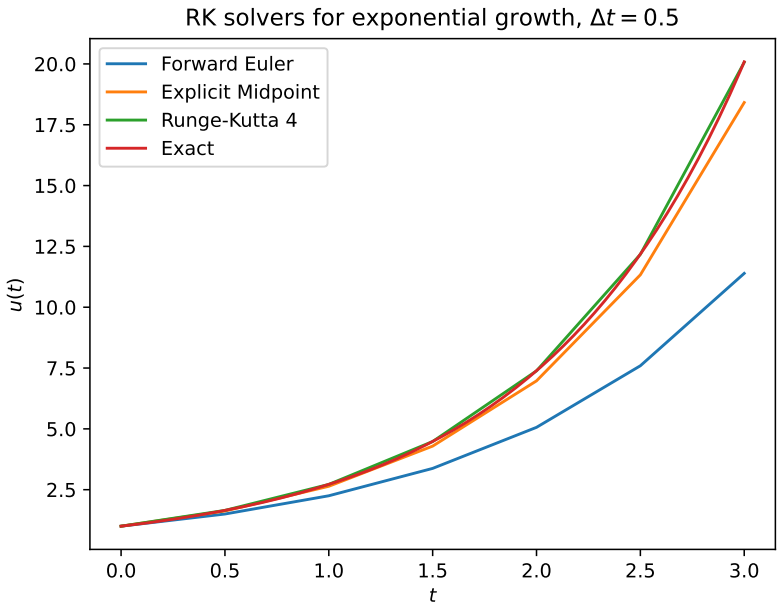

Fig. 2.1 Numerical solutions of the exponential growth problem, computed with `ForwardEuler`, `ImplicitMidpoint` and `RungeKutta4`. All the solvers use $\Delta t = 0.5$, to highlight the difference in accuracy.

2.3 Testing the solvers

In Chapter 1 we showed how to compute the error in the numerical solution, and in particular how we could verify that the error behaved as predicted by the theoretical convergence of the applied solvers. Such tests are extremely valuable for verifying that we have implemented the ODE solvers correctly, and can easily be extended to the higher order solvers. As an example, the following code defines a dictionary containing three different solver classes and their theoretical order, and solves the simple exponential ODE for all three solvers.

```
from ODESolver import *
import numpy as np


def rhs(t, u):
    return u


def exact(t):
    return np.exp(t)

solver_classes = [(ForwardEuler,1), (Heun,2),
                  (ExplicitMidpoint,2), (RungeKutta4,4)]

for solver_class, order in solver_classes:
    solver = solver_class(rhs)
    solver.set_initial_condition(1.0)

    T = 3.0
    t_span = (0, T)
    N = 30
    print(f'{solver_class.__name__}, order = {order}')
    print(f'Time step (dt)   Error (e)     e/dt**{order}')
    for _ in range(10):
        t, u = solver.solve(t_span, N)
        dt = T / N
        e = abs(u[-1] - exact(T))
        if e < 1e-13:  # break if error is close to machine precision
            break
        print(f'{dt:<14.7f}   {e:<12.7f}  {e/dt**order:5.4f}')
        N = N * 2
```

The code is nearly identical to the FE convergence test in Section 1.5, except that we loop over a list of tuples that contain the four method classes and their corresponding order. The output is also nearly identical to the previous version, but repeated for all four solvers, and we use the built-in class attribute `__name__` to extract and print the name of each solver. Three columns are written to the screen, containing, respectively, the time step Δt, the error e at time $t = 3.0$, and finally $e/\Delta t^p$, where p is the order of the method. For the two first methods the output is exactly as expected, and

it is therefore not shown here. The numbers in the rightmost column are approximately constant, confirming that the error is in fact proportional to Δt^p. However, the last part of the output, for the forth order Runge-Kutta method, looks like this:

```
RungeKutta4 order = 4
Time step (dt)   Error (e)     e/dt**4
0.1000000        0.0000462     0.4620
0.0500000        0.0000030     0.4817
0.0250000        0.0000002     0.4918
0.0125000        0.0000000     0.4969
0.0062500        0.0000000     0.4995
0.0031250        0.0000000     0.5006
0.0015625        0.0000000     0.5025
0.0007813        0.0000000     0.5436
0.0003906        0.0000000     5.1880
0.0001953        0.0000000     102.5391
```

We see that the $e/\Delta t^p$ numbers are close to constant for a while, in accordance with the convergence order of the method, but then increase for the smallest values of Δt. This behavior is not uncommon to observe in convergence tests like this, in particular for high-order methods, and it is caused by the finite accuracy of number representation on a computer. As the numerical errors become smaller and approach the machine precision ($\approx 10^{-16}$), roundoff error start to dominate the overall error and convergence is lost.

There are many alternative ways to check the implementation of ODE solvers. One option is to consider an even simpler ODE, where the right hand side is a constant, i.e., $u'(t) = f(t,u) = C$. The solution to this simple ODE is of course $u(t) = Ct + u_0$, where u_0 is the initial condition. All the numerical methods considered in this book will capture this solution to machine precision, and we can write a general test function which takes advantage of this:

```
def test_exact_numerical_solution():
    solver_classes = [ForwardEuler, Heun,
                      ExplicitMidpoint, RungeKutta4]
    a = 0.2
    b = 3

    def f(t, u):
        return a

    def u_exact(t):
        """Exact u(t) corresponding to f above."""
        return a * t + b

    u0 = u_exact(0)
    T = 8
    N = 10
    tol = 1E-14
    t_span = (0, T)
```

```
    for solver_class in solver_classes:
        solver = solver_class(f)
        solver.set_initial_condition(u0)
        t, u = solver.solve(t_span, N)
        u_e = u_exact(t)
        max_error = abs((u_e - u)).max()
        msg = f'{solver_class.__name__} failed, error={max_error}'
        assert max_error < tol, msg
```

Similar to the convergence check illustrated below, this code will loop through all the solver classes, solve the simple ODE, and check that the resulting error is within the tolerance.

Both of the methods shown here to verify the implementation of our solvers have some limitations. The most important one is that they both solve very simple ODEs, and it is entirely possible to introduce errors in the code that will only present themselves for more complex problems. However, they have the advantage of being simple and completely general, and can easily be applied to any newly implemented ODE solver class. Many common implementation errors, for instance getting a single parameter wrong in Runge-Kutta method, will often show up even for this simple problems. They can therefore provide a good initial indication that the implementation is correct, which can be followed by more extensive tests if needed.

Chapter 3
Stable solvers for stiff ODE systems

In the previous chapter we introduced explicit Runge-Kutta (ERK) methods and observed how they could conveniently be implemented as a hierarchy of Python classes. For most ODE systems, replacing the simple forward Euler method with a higher-order ERK method will significantly reduce the number of time steps needed to reach a specified accuracy. In most cases it will also lead to a reduced overall computation time, since the additional cost for each time step is more than outweighed by the reduced number of steps. However, for a certain class of ODEs we may observe that all the ERK methods require very small time steps, and any attempt to increase the time step will lead to spurious oscillations and possible divergence of the solution. These ODE systems are usually referred to as *stiff*, and none of the explicit methods introduced in the previous chapters do a very good job at solving them. We shall see that implicit solvers such as implicit Runge-Kutta (IRK) methods are far better suited for stiff problems, and may give substantial reduction of the computation time for challenging problems.

3.1 Stiff ODE systems and stability

One very famous example of a stiff ODE system is the Van der Pol equation, which can be written as an initial value problem on the form

$$y_1' = y_2, \qquad y_1(0) = 1, \tag{3.1}$$

$$y_2' = \mu(1 - y_1^2)y_2 - y_1, \quad y_2(0) = 0. \tag{3.2}$$

The parameter μ is a constant which determines the properties of the system, including its "stiffness". For $\mu = 0$ the problem is a simple oscillator with analytical solution $y_1 = \cos(t), y_2 = \sin(t)$, while for non-zero values of μ the solution shows far more complex behavior. The following code implements

this system and solves it with the `ForwardEuler` subclass of the `ODESolver` class hierarchy.

```
from ODESolver import *
import numpy as np
import matplotlib.pyplot as plt

class VanderPol:
    def __init__(self,mu):
        self.mu = mu

    def __call__(self,t,u):
        du1 = u[1]
        du2 = self.mu*(1-u[0]**2)*u[1]-u[0]
        return du1,du2

model = VanderPol(mu=1)

solver = ForwardEuler(model)
solver.set_initial_condition([1,0])

t,u  = solver.solve(t_span=(0,20),N=1000)

plt.plot(t,u)
plt.show()
```

Figure 3.1 shows the solutions for $\mu = 0, 1$ and 5. Setting μ even higher, for instance $\mu = 50$, leads to a divergent (unstable) solution with the time step used here ($\Delta t = 0.02$). Replacing the FE method with one of the more accurate ERK method may help a little, but not much. It does help to reduce the time step dramatically, but the resulting computation time may be substantial. The time step for this problem is dictated by stability requirements rather than our desired accuracy, and there may be significant gains from choosing a solver that is more stable than the ERK methods considered so far.

Before introducing more stable solvers, it is useful to examine the observed stability problems in a bit more detail. Why does the solution of the Van der Pol model fail so badly for large values of μ? And, more generally, what are the properties of an ODE system that makes it stiff? To answer these questions, it is useful to start with a simpler problem than the Van der Pol model. Consider, for instance, a simple IVP known as the Dahlquist test equation;

$$u' = \lambda u, \quad u(0) = 1, \tag{3.3}$$

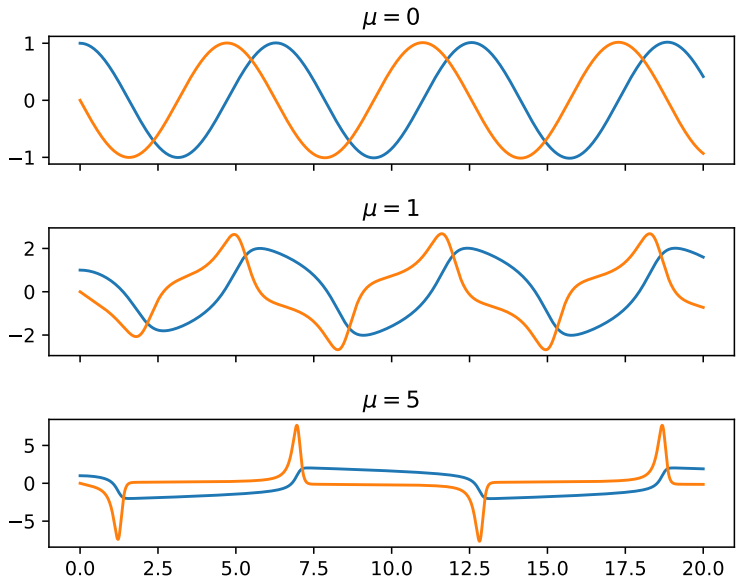

Fig. 3.1 Solutions of the Van der Pol model for different values of μ.

where λ may be a complex number.[1] With $\lambda = 1$ this is the simple exponential growth problem considered earlier, but in this chapter we are primarily interested in λ with negative real part, i.e., either real or complex λ that satisfy $\Re(\lambda) < 0$. In such cases the solution of (3.3) decays over time and is completely stable, but we shall see that the numerical solutions do not always retain this property.

Following the definition in [1], we say that problem (3.3) is stiff for an interval $[0, b]$ if the real part of λ satisfies

$$b\Re(\lambda) \ll -1.$$

For more general non-linear problems, such as the Van der Pol model in (3.1)-(3.2), the system's stiffness is characterized by the eigenvalues λ_i of the local Jacobian matrix J of the right-hand side function f. The Jacobian is defined by

$$J_{ij} = \frac{\partial f_i(t, y)}{\partial y_j},$$

[1] Note that the implementation of the solvers in this book does not support solving this ODE for complex λ. Allowing complex values in the stability analysis is still relevant, since for systems of ODEs the relevant values are the *eigenvalues* of the right-hand side, and these may be complex.

and the problem is stiff for an interval $[0, b]$ if

$$b \min_i \Re(\lambda_i) \ll -1.$$

These definitions show that the stiffness of a problem is not only a function of the ODE itself, but also of the length of the solution interval (b), which may be a bit surprising. We can get an understanding of why the interval of interest is important by looking at (3.3). If λ is large and negative, we need to choose a small Δt to maintain stability of explicit solvers, as will be discussed in detail below. However, if we are only interested in solving the equation over a very small time interval, i.e., b is small, using a small Δt is not really a problem, and by the definition above the problem will no longer be stiff. In the ODE literature one will also find more pragmatic definitions of stiffness, for instance that an equation is stiff if the time step needed to maintain stability of an explicit method is much smaller than the time step dictated by the accuracy requirements [1, 2]. A detailed discussion of stiff ODE systems can be found in, for instance, [1, 9].

Eq. (3.3) is the foundation for *linear stability analysis*, which is a very useful technique for analyzing and understanding the stability of ODE solvers. The solution to the equation is $u(t) = e^{\lambda t}$, which obviously grows very rapidly if λ has a positive real part. We are therefore primarily interested in the case $\Re(\lambda) < 0$, for which the analytical solution is stable, but our choice of solver may introduce *numerical instabilities*. The Forward Euler method applied to (3.3) gives the update formula

$$u_{n+1} = u_n + \Delta t \lambda u_n = u_n(1 + \Delta t \lambda),$$

and for the first step, since we have $u(0) = 1$, we have

$$u_1 = 1 + \Delta t \lambda. \tag{3.4}$$

The analytical solution decays exponentially for $\Re(\lambda) < 0$, and it is natural to require the same behavior of the numerical solution, and this gives the requirement that $|1 + \Delta t \lambda| \leq 1$. If λ is real and negative, the time step must be chosen to satisfy $\Delta t \leq -2/\lambda$ to ensure stability. Keep in mind that this criterion does not necessarily give a very accurate solution, and it may even oscillate and look completely different from the exact solution. However, choosing Δt to satisfy the stability criterion ensures that the solution, as well as any spurious oscillations and other numerical artefacts, decay with time.

We have observed that the right-hand side of (3.4) contains critical information about the stability of the FE method. This expression is often called the *stability function* or the *amplification factor* of the method, and is written on the general form

$$R(z) = 1 + z.$$

The FE method is stable for all values $\lambda\Delta t$ which give $|R(\lambda\Delta t)| < 1$, and this region of $\lambda\Delta t$ values in the complex plane is referred to as the method's *region of absolute stability*, or simply its *stability region*. The stability region for the FE method is shown in the left panel of Figure 3.2. The stability domain is a circle with center $(-1, 1)$ and radius one. Obviously, if $\lambda \ll 0$, require $\lambda\Delta t$ to lie within this circle is quite restrictive for the choice of Δt.

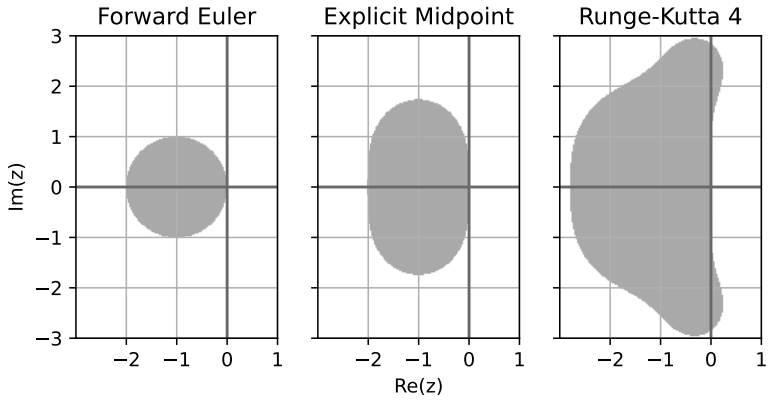

Fig. 3.2 Stability regions for explicit Runge-Kutta methods. From left: forward Euler, explicit midpoint, and the fourth order method given by (2.10)-(2.14).

We can easily extend the linear stability analysis to the other explicit RK methods introduced in Chapter 2. For instance, applying a single step of the explicit midpoint method given by (2.2)-(2.4) to (3.3) gives

$$u(\Delta t) = 1 + \lambda\Delta t + \frac{(\Delta t\lambda)^2}{2},$$

and we identify the stability function for this method as

$$R(z) = 1 + z + \frac{z^2}{2}.$$

The corresponding stability region is shown in the middle panel of Figure 3.2. For the fourth order RK method defined in (2.10)-(2.14), the same steps reveal

that the stability function is

$$R(z) = 1 + z + \frac{z^2}{2} + \frac{z^3}{6} + \frac{z^4}{24},$$

and the stability region is shown in the right panel of Figure 3.2. We observe that the stability regions of the two higher-order RK methods are larger than that of the FE method, but not much. In fact, if we consider the computational cost of each time step for these methods, the FE method is usually superior for problems where the time step is governed by stability.

It can be shown that the stability function for an s-stage explicit RK method is always a polynomial of degree $\leq s$, and it can easily be verified that the stability region defined by such a polynomial will never be very large. To obtain a significant improvement of this situation, we typically need to replace the explicit methods considered so far with implicit RK methods.

3.2 Implicit methods for stability

Since (3.3) is stable for all values of λ with a negative real part, it is natural to look for numerical methods with the same property. This means that the stability domain for the method covers the entire left half of the complex plane, or that its stability function $|R(z)| \leq 1$ whenever $\Re(z) \leq 0$. This property is called *A-stability*. As noted above, the stability function of an explicit RK method is always a polynomial, and no polynomial satisfies $|R(z)| < 1$ for all $z < 0$. Therefore, there are no A-stable explicit RK methods. An even stronger stability requirement can be motivated by the fact that for $\lambda \ll 0$, the solution to (3.3) decays very rapidly. It is natural to expect the same behavior of the numerical solution, by requiring $|R(z)| \to 0$ as $z \to -\infty$. This property is referred to as *stiff decay*, and an A-stable method that also has stiff decay is called an *L-stable* method, see, for instance, [1, 9] for more details.

The simplest implicit RK method is the backward Euler (BE) method, which can be derived in exactly the same way as the FE method, by approximating the derivative with a simple finite difference. The only difference from the FE method is that the right-hand side is evaluated at step $n+1$ rather than step n. For a general ODE, we have

$$\frac{u_{n+1} - u_n}{\Delta t} = f(t_{n+1}, u_{n+1}),$$

and if we rearrange the terms we get

$$u_{n+1} - \Delta t f(t_{n+1}, u_{n+1}) = u_n. \tag{3.5}$$

Although the derivation is very similar to the FE method, there is a fundamental difference in that the unknown u_{n+1} occurs as an argument in the

right-hand side function $f(t,u)$. Therefore, for nonlinear f, (3.5) is a nonlinear algebraic equation that must be solved for the unknown u_{n+1}, instead of the explicit update formula we had for the FE method. This requirement makes implicit methods more complex to implement than explicit methods, and they tend to require far more computations per time step. Still, as we will demonstrate later, the superior stability properties still make implicit solvers better suited for stiff problems.

We will consider the implementation of implicit solvers in Section 3.3 below, but let us first study the stability of the BE method and other implicit RK solvers using the linear stability analysis introduced above. Applying the BE method to (3.3) yields

$$u_{n+1}(1-\Delta t\lambda) = u_n,$$

and for the first time step, with $u(0)=1$, we get

$$u_1 = \frac{1}{1-\Delta t\lambda}.$$

The stability function of the BE method is therefore $R(z) = 1/(1-z)$, and the corresponding stability domain is shown in the left panel of Figure 3.3. The method is stable for all choices of $\lambda\Delta t$ *outside* the circle with radius one and center at (1,0) in the complex plane, confirming that the BE is a very stable method. It is A-stable, since the stability domain covers the entire left half of the complex plane, and it is also L-stable since the stability function satisfies $R(z)\to 0$ as $\Re(z)\to-\infty$.

The BE method fits into the general RK framework defined by (2.8)-(2.9) in Chapter 2, with a single stage ($s=1$), and $a_{11}=b_1=c_1=1$. As for the FE method considered in Chapter 2, we can reformulate the method slightly to introduce a stage derivative and make it obvious that the BE method is part of the RK family:

$$k_1 = f(t_n+\Delta t, u_n+\Delta t k_1), \tag{3.6}$$
$$u_{n+1} = u_n + \Delta t k_1. \tag{3.7}$$

The explicit midpoint and trapezoidal methods mentioned above also have their implicit counterparts. The implicit midpoint method is given by idxmidpoint method !implicit

$$k_1 = f(t_n+\Delta t/2, u_n+k_1\Delta t/2), \tag{3.8}$$
$$u_{n+1} = u_n + \Delta t k_1, \tag{3.9}$$

while the implicit trapezoidal rule, or Crank-Nicolson method, is given by

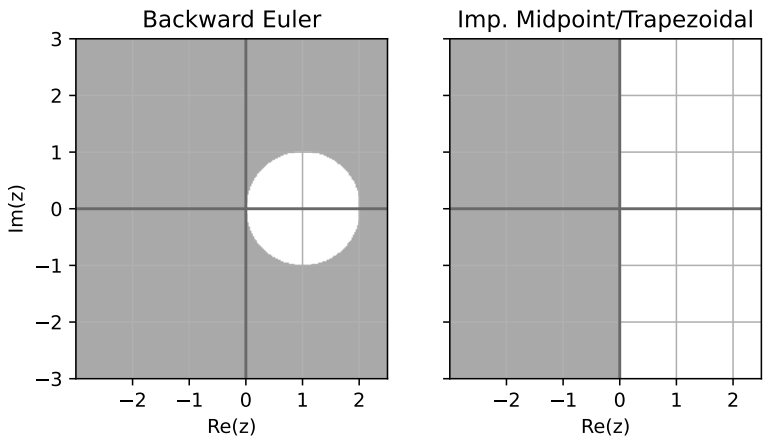

Fig. 3.3 Stability regions for the backward Euler method (left) and the implicit midpoint method and trapezoidal method (right).

$$k_1 = f(t_n, u_n), \tag{3.10}$$

$$k_2 = f(t_n + \Delta t, u_n + \Delta t k_2), \tag{3.11}$$

$$u_{n+1} = u_n + \frac{\Delta t}{2}(k_1 + k_2). \tag{3.12}$$

Note that this formulation of the Crank-Nicolson is not very common, and can be simplified considerably by eliminating the stage derivatives and defining the method in terms of u_n and u_{n+1}. However, the formulation in (3.10)-(3.12) is convenient for highlighting that it is in fact an implicit RK method. The implicit nature of the simple methods above is apparent from the formulas; one of the stage derivatives must be computed by solving an equation involving the non-linear function f rather than from an explicit update formula. The Butcher tableaus of the three methods are given by

$$\begin{array}{c|c} 1 & 1 \\ \hline & 1 \end{array}, \qquad \begin{array}{c|c} 1/2 & 1/2 \\ \hline & 1 \end{array}, \qquad \begin{array}{c|cc} 0 & & \\ 1 & 0 & 1 \\ \hline & 1/2 & 1/2 \end{array}, \tag{3.13}$$

from left to right for backward Euler, implicit midpoint and the implicit trapezoidal method.

The implicit midpoint method is and the implicit trapezoidal method have the same stability function, given by $R(z) = (2+z)/(2-z)$. The corresponding stability domain covers the entire left half-plane of the complex plane, shown in the right panel of Figure 3.3. Both the implicit midpoint method and the trapezoidal method are therefore A-stable methods. However, we have $R(z) \to 1$ as $z \to -\infty$, so the methods do not have stiff decay and are therefore not L-stable. In general, the stability functions of implicit RK methods are always rational functions, i.e., given by

$$R(z) = \frac{P(z)}{Q(z)},$$

where P, Q are polynomials of degree at most s. (Recall from Section 3.1 above that the stability functions for the explicit methods are always polynomials of degree at most s.)

The accuracy of the implicit methods considered above can easily be calculated using a Taylor series expansion as outlined in Section 1.5, and confirms that the backward Euler method is first order accurate while the two others are second order methods. We mentioned above that an explicit Runge-Kutta method with s stages has order $p \leq s$, but with implicit methods we have greater freedom in choosing the coefficients a_{ij} and therefore potentially higher accuracy for a given number of stages. In fact, the maximum order for an implicit RK method is $p = 2s$, which is precisely the case for the implicit midpoint method, having $s = 1$ and $p = 2$. We will consider more advanced implicit RK methods later, but let us first have a look at how we can implement the methods introduced so far.

3.3 Implementing implicit Runge-Kutta methods

In the previous section we have demonstrated the superior stability of the implicit methods, and also mentioned that the accuracy is generally higher, for a fixed number of stages. So why are not IRK solvers the natural choice for all ODE problem? The answer to this question is of course the fact that they are implicit, so the stage derivatives are defined in terms of non-linear equations rather than explicit formulae. This fact complicates the implementation of the methods and makes each time step far more computationally expensive. Because of the latter, explicit solvers are usually more efficient for all non-stiff problems, and implicit solvers should only be used for stiff ODEs.

For scalar ODEs, solving an equation such as (3.5) or (3.8) with Newton's method is usually not very challenging. However, we are most interested in solving systems of ODEs, which complicates the task a bit, since we then need to solve a system of coupled non-linear equations. Applying Newton's

method to a system of equations requires the solution of a system of linear equations for every iteration, and we are left with a wide variety of choices for how to solve these, as well as other solver choices and parameters that may be tuned to optimize the performance. In the present text we focus on understanding the fundamental ideas of the IRK solvers, and we will not dive into the details of optimizing the performance. We will therefore base our implementation on built-in equation solvers from SciPy. We will start with the backward Euler method, being the simplest, but we will keep the implementation sufficiently general to be easily extended to more advanced implicit methods. The interested reader may refer to, for instance, [1,9] for a detailed discussion of solver optimization and choices to improve the computational performance.

If we examine the `ODESolver` class first introduced in Chapter 2, we may observe that many of the administrative tasks involved in the RK methods are the same for implicit as for explicit methods. In particular, the initialization of the solution arrays and the for-loop that advances the solution are exactly the same, but advancing the solution from one step to the next is quite different. It is therefore convenient to implement the implicit solvers as part of the existing class hierarchy, and let the `ODESolver` superclass handle the tasks of initializing the solver as well as the main solver loop. The different explicit methods introduced in Chapter 2 where then realized through different implementations of the `advance` method. We can use the same approach for implicit methods, but since each step involves a few more operations it is convenient to introduce a couple of additional methods. For instance, a compact implementation of the backward Euler method may look as follows:

```
from ODESolver import *
from scipy.optimize import root

class BackwardEuler(ODESolver):

    def stage_eq(self,k):
        u, f, n, t = self.u, self.f, self.n, self.t
        dt = self.dt
        return k - f(t[n]+dt,u[n]+dt*k)

    def solve_stage(self):
        u, f, n, t = self.u, self.f, self.n, self.t
        k0 = f(t[n],u[n])
        sol = root(self.stage_eq,k0)
        return sol.x

    def advance(self):
        u, f, n, t = self.u, self.f, self.n, self.t
        dt = self.dt
        k1 = self.solve_stage()
        return u[n]+dt*k1
```

Compared with the explicit solvers presented in Chapter 2 we have introduced two additional methods in our `BackwardEuler` class. The first of these, `stage_eq(self,k)`, is simply a Python implementation of (3.6), which defines the non-linear equation for the stage derivative. The method takes the stage derivative `k` as input and returns the residual of (3.6), which makes it suitable for use with SciPy non-linear equation solvers. The actual solution of the stage derivate equation is handled in the `solve_stage` method, which first computes an initial guess `k0` for the stage derivative, and then passes this guess and the function `stage_eq` to SciPy's `root` function for solving the equation. The `root` function is a general tool for solving non-linear equations on the form $g(x) = 0$, which we apply to solve the stage equation $k_1 - f(t_n + \Delta t, u_n + \Delta t k_1) = 0$. It returns an object of the class `OptimizeResult`, which includes the solution as an attribute `x`, as well as numerous other attributes containing information about the solution process. We refer to the SciPy documentation for further details on the `OptimizeResult` and the `root` function.

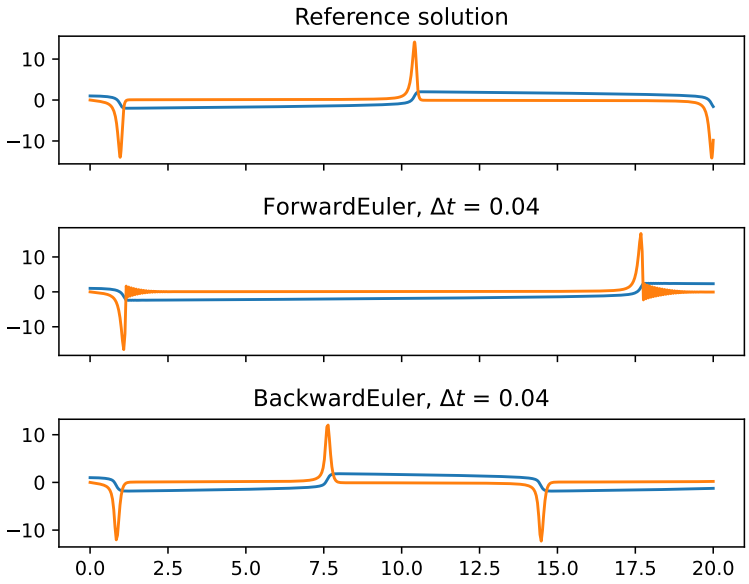

Fig. 3.4 Solutions of the Van der Pol model for $\mu = 10$, using the forward and backward Euler methods with $\Delta t = 0.04$.

We can demonstrate the superior stability of the backward Euler method by returning to the Van der Pol equation considered above. Setting, for instance, $\mu = 10$, and solving the model with both the forward and backward

Euler method gives the plots shown in Figure 3.4. The top panel shows a reference solution computed with the SciPy `solve_ivp` solver and very low tolerance (`rtol=1e-10`). The middle panel shows the solution produced by forward Euler with $\Delta t = 0.04$, showing visible oscillations in one of the solution components. Increasing the time step further leads to a divergent solution. The lower panel shows the solution from backward Euler with $\Delta t = 0.04$, which is obviously a lot more stable, but still quite different from the reference solution in the top panel. With the backward Euler method, increasing the time step further will still give a stable solution, but it does not look like the exact solution at all. This little experiment illustrates the need to consider both accuracy and stability when solving challenging ODEs systems.[2]

Just as we did for the explicit methods in Chapter 2, it is possible to reuse code from the `BackwardEuler` class to implement other solvers. Extensive code reuse for a large group of implicit solvers requires a small rewrite of the code above to a more general form, which will be presented in the next section. However, we may observe that a simple solver like the Crank-Nicolson method can be realized as a very small modification of our `BackwardEuler` class. A class implementation may look like

```
class CrankNicolson(BackwardEuler):
    def advance(self):
        u, f, n, t = self.u, self.f, self.n, self.t
        dt = self.dt
        k1 = f(t[n],u[n])
        k2 = self.solve_stage()
        return u[n]+dt/2*(k1+k2)
```

Here, we utilize the fact that the stage k_1 in the Crank-Nicolson is explicit and does not require solving an equation, while the definition of k_2 is identical to the definition of k_1 in the backward Euler method. We can therefore reuse both the `stage_eq` and `solve_stage` methods directly and only the `advance` method needs to be reimplemented. This compact implementation of the Crank-Nicolson method is convenient for code reuse, but it may be argued that it violates a common principle of object-oriented programming. Subclassing and inheritance is considered an "is-a" relationship, so this class implementation implies that an instance of the `Crank-Nicolson` class is also an instance of the `BackwardEuler` class. While this works fine in the program, and is convenient for code reuse, it is not a correct representation of the relationship between the two numerical methods. The Crank-Nicolson method is not a special case of the backward Euler, but, as noted above, both methods belong to the group of implicit RK solvers. In the following sections we will describe an alternative class hierarchy which is based on this relationship, and

[2]Note that the accompanying source code for the book includes a script to produce Figure 3.4, as well as many of the other figures in the book. It is a good exercise to run these scripts on your own and adjust the time step and other parameters, to get a better understanding of how the solvers work.

enables a compact implementation of RK methods by utilizing the general formulation in (2.8)-(2.9).

3.4 Implicit methods of higher order

Just as for the ERK methods considered in Chapter 2, the accuracy of IRK methods can be increased by adding more stages. However, for implicit methods we have even more freedom in choosing the parameters a_{ij}, and the choice of these impacts both the accuracy and the computational complexity of the methods. We will here consider two main branches of IRK methods; the so-called *fully implicit* methods and the *diagonally implicit* methods. Both classes of methods are quite popular and commonly used, and both have their advantages and drawbacks.

3.4.1 Fully implicit RK methods

The most general form of RK methods are the fully implicit methods, often referred to as FIRK methods. These solvers are simply defined by (2.8)-(2.9), with all coefficients a_{ij} (potentially) non-zero. For a method with more than one stage, this formulation implies that all stage derivatives depend on all other stage derivatives, so we need to determine them all at once by solving a single system of non-linear equations. This operation is quite expensive, but the reward is that the FIRK methods have superior stability and accuracy for a given number of stages. A FIRK method with s stages can have order at most $2s$, which was the case for the implicit midpoint method in (3.8)-(3.9).

Many of the most popular FIRK methods are based on combining standard quadrature methods for numerical integration with the idea of *collocation*. We present the basic idea of the derivation here, since many important methods are based on the same foundation. For the complete details we refer to, for instance, [9]. Recall from Chapter 2 that all Runge-Kutta methods can be viewed as approximations of (2.1), where the integral is approximated by a weighted sum. We set

$$u(t_{n+1}) = u(t_n) + \int_{t_n}^{t_{n+1}} f(t, u(t)) \approx u(t_n) + \sum_{i=1}^{s} b_i k_i,$$

where b_i are the weights and k_i are the stage derivatives, which could be interpreted as approximations of the right-hand side function $f(t, u)$ at distinct time points $t_n + \Delta t c_i$.

Numerical integration is a very well studied branch of numerical analysis, and it is natural to choose the integration points c_i and weights b_i based on

standard quadrature rules with well established properties. Such quadrature rules are often derived by approximating the integrand with a polynomial which interpolates the function f in distinct points, and then integrating the polynomial exactly. The same idea can be used in the derivation of implicit RK methods. We approximate the solution u on the interval $t_n < t \leq t_{n+1}$ by a polynomial $P(t)$ of degree at most s, and then require that $P(t)$ solves the ODE exactly in distinct points $t_n + c_i \Delta t$. This requirement, i.e., that

$$P'(t_i) = f(t_i, P(t_i)), \quad t_i = t_n + c_i \Delta t, i = 1, \ldots, s. \tag{3.14}$$

is known as collocation, and is a widely used idea in numerical analysis. It can be shown that, given a choice of quadrature points c_i, the collocation equations given by (3.14) determines the remaining method coefficients a_{ij} and b_i uniquely, see [1] for details.

A convenient way to derive FIRK methods is to choose a set of collocation points c_i, typically chosen from standard quadrature rules, and solve (3.14) to determine the remaining parameters. This approach has led to families of FIRK methods based on common rules for numerical integration. For instance, choosing c_i as Gauss points gives rise to the Gauss methods, which are the most accurate methods for a given number of stages, having order $2s$. The single stage Gauss method is the implicit midpoint method introduced above, while the fourth order Gauss method with $s = 2$ is defined by the Butcher tableau

$$\begin{array}{c|cc} \frac{3-\sqrt{3}}{6} & \frac{1}{4} & \frac{3-2\sqrt{3}}{12} \\ \frac{3+\sqrt{3}}{6} & \frac{3+2\sqrt{3}}{12} & \frac{1}{4} \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}.$$

The Gauss methods are A-stable but not L-stable, and since we typically only use FIRK methods for challenging stiff problems where stability is important, another family of FIRK methods is more popular in practice. These are the Radau IIA methods, which are based on Radau quadrature points which include the right end of the integration interval (i.e., $c_s = 1$). The one-stage Radau IIA method is the backward Euler method, while two- and three-stage versions are given by

$$\begin{array}{c|cc} 1/3 & 5/12 & -1/12 \\ 1 & 3/4 & 1/4 \\ \hline & 2/3 & 1/4 \end{array}, \qquad \begin{array}{c|ccc} \frac{4-\sqrt{6}}{10} & \frac{88-7\sqrt{6}}{360} & \frac{296-169\sqrt{6}}{1800} & \frac{-2+3\sqrt{6}}{225} \\ \frac{4+\sqrt{6}}{10} & \frac{296+169\sqrt{6}}{1800} & \frac{88+7\sqrt{6}}{360} & \frac{-2-3\sqrt{6}}{225} \\ 1 & \frac{16-\sqrt{6}}{36} & \frac{16+\sqrt{6}}{36} & \frac{1}{9} \\ \hline & \frac{16-\sqrt{6}}{36} & \frac{16+\sqrt{6}}{36} & \frac{1}{9} \end{array}.$$

The Radau IIA methods have order $2s-1$, and their stability functions are $(s-1, s)$ Padé approximations to the exponential function, see [9] for details. For the two- and three-stage methods above, the stability functions are

$$R(z) = \frac{1 + z/3}{1 - 2z/3 + z^2/6},$$
$$R(z) = \frac{1 + 2z/5 + z^2/20}{1 - 3z/5 + 3z^2/20 - z^2/60},$$

respectively, with stability domains shown in Figure 3.5. The methods are L-stable, which makes them a popular choice for solving stiff ODE systems. However, as noted above, the fact that all $a_{ij} \neq 0$ complicates the implementation of the methods and makes each time step computationally expensive. All the s equations of (2.8) become fully coupled and need to be solved simultaneously. For an ODE system consisting of m ODEs, we need to solve a system of ms non-linear equations for each time step. We will come back the implementation of FIRK methods in Section 3.5, but let us first introduce a slightly simpler class of implicit RK solvers.

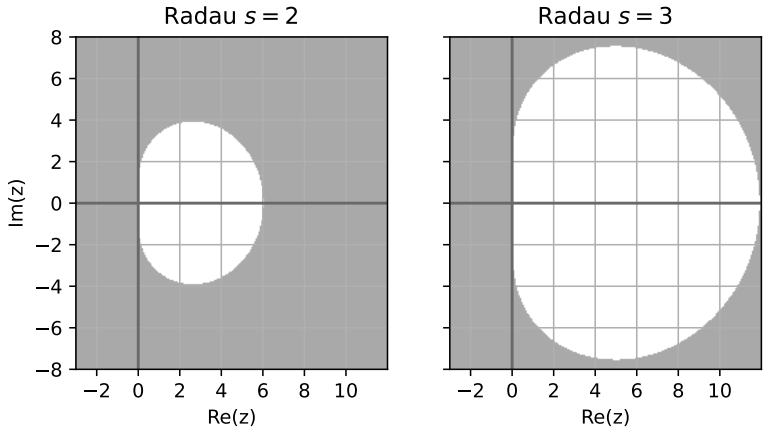

Fig. 3.5 The shaded area is the stability region for two of the RadauIIA methods, with $s = 2$ (left) and $s = 3$ (right).

3.4.2 Diagonally implicit RK methods

Diagonally implicit RK methods, or DIRK methods for short, are also sometimes also referred to as semi-explicit methods. For these methods, we have $a_{ij} = 0$ for all $j > i$. (Notice the small but important difference from the explicit methods, where we have $a_{ij} = 0$ for $j \geq i$.) The consequence of this choice is that the equation for a single stage derivative k_i does not involve stages k_{i+1}, k_{i+2}, and so forth, and we can therefore solve for the stage derivatives one by one sequentially. We still need to solve non-linear equations to determine each k_i, but we can solve s systems of m equations rather than one large system to compute all the stages at once. This property simplifies the implementation and reduces the computational expense per time step. However, as expected, the restriction on the method coefficients reduces the accuracy and stability compared with the FIRK methods. A general DIRK method with s stages has maximum order $s+1$, and methods optimized for stability typically have even lower order.

From the definition of the DIRK methods, we may observe that the implicit midpoint method introduced above is, technically, a DIRK method. However, this method is also a fully implicit Gauss method, and is not commonly referred to as a DIRK method. The distinction between FIRK and DIRK methods is only meaningful for $s > 1$. The Crank-Nicolson (implicit trapezoidal) method given by (3.10)-(3.12) is also a DIRK method, which is evident from the rightmost Butcher tableau in (3.13). These methods are, however, only A-stable, and it is possible to derive DIRK methods with better stability properties. An example of an L-stable, two-stage DIRK method of order two is given by

$$\begin{array}{c|cc} \gamma & \gamma & 0 \\ 1 & 1-\gamma & \gamma \\ \hline & 1-\gamma & \gamma \end{array}, \tag{3.15}$$

with stability function

$$R(z) = \frac{1+z(1-2\gamma)}{(1-z\gamma)^2}.$$

The method is A-stable for $\gamma > 1/4$, and for $\gamma = 1 \pm \sqrt{2}/2$ the method is L-stable and second order accurate. Note that choosing $\gamma > 1$ means that we estimate the stage derivatives outside the interval (t_n, t_{n+1}), and for the last step outside the time interval of interest. While this does not affect the stability or accuracy of the method it may not make sense for all ODE problems, and the most popular choice is therefore $\gamma = 1 - \sqrt{2}/2$ (≈ 0.293). Notice also that in this method the two diagonal entries of a_{ij} are identical, we have $a_{11} = a_{22} = \gamma$. This choice is very common in DIRK methods, and methods of this kind are usually referred to as *singly diagonally implicit* RK (SDIRK) methods. The main benefit of this structure is that the non-linear equations for each stage derivative become very similar, which can be utilized when solving the equations with quasi-Newton methods. This

benefit may not be very obvious for the examples in this book, since we rely on the generic `root` function from `scipy.optimize` to solve all the non-linear equations. However, if we wanted to improve the computational performance of the solvers, a natural place to start would be to implement our own quasi-Newton solver, which takes advantage of the particular structure of the non-linear equations. We will not go into the details of such an implementation here, but it is worth commenting on some aspects in order to understand the popularity of the SDIRK methods. The central point is that when applying Newton's method to solve a general non-linear system $g(u) = 0$, each iteration involves solving linear systems on the form $J_g \Delta u = -g(u^k)$, where Δu is the increment to the solution, u^k is the solution value at the previous iteration, and J_g is the Jacobian matrix of g, defined by

$$J_g = \frac{\partial g_i}{\partial u_j}.$$

For a general DIRK method, the non-linear equation to compute stage derivative k_i is given by

$$k_i = f(t_n + c_i \Delta t, u_n + \Delta \sum_{j=1}^{i} a_{ij} k_j),$$

which we can write on the form $g(k_i) = 0$, with

$$g(k_i) = k_i - f(t_n + c_i \Delta t, u_n + \Delta \left(\sum_{j=1}^{i-1} a_{ij} k_j + a_{ii} k_i \right)).$$

Note that we have split the sum over the stage derivatives, to highlight the fact that when solving for k_i, the values k_j for $j < i$ are known. The Jacobian J_g is found by differentiating g with respect to k_i, to get

$$J_g = I - \Delta t a_{ii} J_f,$$

where J_f is the Jacobian of the right-hand side function f. We observe that if we have $a_{ii} = \gamma$ for all stages, the Jacobian matrices will also be identical, which can be utilized to optimize the solution of the linear systems. We refer to, for instance, [9] for a detailed overview of solving non-linear equations arising in SDIRK mehtods.

While we do not aim to present a complete overview of the various subcategories of RK methods, one additional class of method is worth mentioning. These are the so called ESDIRK methods, which are simply SDIRK methods where the first stage is explicit. The motivation for such methods is that the non-linear algebraic equations involved in the implicit methods are always solved with iterative methods, which require an initial guess for the solution. For SDIRK methods, it is convenient to use the previous stage derivate

as initial guess for the next one, which will usually provide a good initial guess. This approach is obviously not possible for the first stage, but an explicit formula for the first stage solves this problem. The simplest ESDIRK method is the implicit trapezoidal (Crank-Nicolson) method introduced above, and a popular extension of this method is given by

$$\begin{array}{r|l} 0 & 0 \\ 2\gamma & \gamma\ \gamma\ 0 \\ 1 & \beta\ \beta\ \gamma \\ \hline & \beta\ \beta\ \gamma \end{array}, \tag{3.16}$$

with $\gamma = 1 - \sqrt{2}/2$ and $\beta = \sqrt{2}/4$. The resulting equations for each time step are

$$\begin{aligned} k_1 &= f(t_n, u_n), \\ k_2 &= f(t_n + 2\gamma\Delta t, u_n + \Delta t(\gamma k_1 + \gamma k_2)), \\ k_3 &= f(t_n + \Delta t, u_n + \Delta t(\beta k_1 + \beta k_2 + \gamma k_3)), \\ u_{n+1} &= u_n + \Delta t(\beta k_1 + \beta k_2 + \gamma k_3). \end{aligned}$$

This method can be interpreted as the sequential application of the trapezoidal method and a popular multistep solver called BDF2 (*backward differentiation formula* of order 2), and it is commonly referred to as the TR-BDF2 method. It is second order accurate, just as the trapezoidal rule, but it is also L-stable and therefore suitable for stiff problems.

3.5 Implementing higher order IRK methods

In Section 3.3 we implemented two of the simplest implicit RK methods by a relatively small extension of the `ODEsolver` class hierarchy. We could easily continue this idea for the more complex IRK methods, and all the different methods could be realized by separate implementations of the three methods `solve_stage`, `stage_eq`, and `advance`. However, these three methods essentially implement the equations given by (2.8)-(2.9), which are common for all RK solvers. It is natural to look for an implementation that allows even more code reuse between the various methods, and we shall see that such a general implementation is indeed possible. However, it still makes sense to treat the fully implicit methods and SDIRK methods separately, since the stage calculations of these two method classes are fundamentally different.

3.5.1 A base class for fully implicit methods

One approach to implement the fully implicit RK methods is to rewrite the `solve_stage`, `stage_eq`, and `advance` methods of the `BackwardEuler` class above in a completely general manner, so they can handle any number of stages and any choice of method parameters a_{ij}, b_i, and c_i. New methods can then be implemented simply by setting the number of stages and defining the parameter values. In the methods considered so far all the method coefficients have been hard-coded into the mathematical expressions, typically inside the `advance` methods, but with the generic approach it is natural to define them as class attributes in the constructor. A general base class for implicit RK methods may look as follows:

```
from ODESolver import *
from scipy.optimize import root

class ImplicitRK(ODESolver):
    def solve_stages(self):
        u, f, n, t = self.u, self.f, self.n, self.t
        s = self.stages
        k0 = f(t[n], u[n])
        k0 = np.tile(k0,s)

        sol = root(self.stage_eq, k0)

        return np.split(sol.x, s)

    def stage_eq(self, k_all):
        a, c = self.a, self.c
        s, neq = self.stages, self.neq

        u, f, n, t = self.u, self.f, self.n, self.t
        dt = self.dt

        res = np.zeros_like(k_all)
        k = np.split(k_all, s)
        for i in range(s):
            fi = f(t[n] + c[i] * dt, u[n] + dt *
                   sum([a[i, j] * k[j] for j in range(s)]))
            res[i * neq:(i + 1) * neq] = k[i] - fi

        return res

    def advance(self):
        b = self.b
        u, n, t = self.u, self.n, self.t
        dt = self.dt
        k = self.solve_stages()

        return u[n] + dt * sum(b_ * k_ for b_, k_ in zip(b, k))
```

Note that we assume that the method parameters are assumed to be held in NumPy arrays `self.a, self.b, self.c`, which need to be defined in subclasses. The `ImplicitRK` class is meant to be a pure base class for holding common code, and is not intended to be a usable solver class in itself. As described in Section 2.2, we could make explicit this abstract nature by using the `abc` module, but for the present text we focus on the fundamentals of the solvers and the class structure, and keep the code as simple and compact as possible.

The three methods are generalizations of the same methods in `BackwardEuler` class, and perform the same tasks, but the abstraction level is higher and the methods rely on a bit of NumPy magic:

- The `solve_stages` method is obviously a generalization of the `solve_stage` method above, and most of the lines are quite similar and should be self-explanatory. However, be aware that we are now implementing a general IRK method with s stages, and we solve a single system of non-linear equations to determine all s stage derivatives at once. The solution of this system is a one-dimensional array of length `self.stages * self.neq`, which contains all the stage derivatives. The line `k0 = np.tile(k0,s)` takes an initial guess `k0` for a single stage, and simply stacks it after itself s times to create the initial guess for all the stages, using NumPy's `tile` function.
- The `stage_eq` method is also a pure generalization of the `BackwardEuler` version, and performs the same tasks. The first few lines should be self-explanatory, while the `res = np.zeros_like(k_all)` defines an array of the correct length to hold the residual of the equation. Then, for convenience, the line `k = np.split(k_all,s)` splits the array `k_all` into a list `k` containing the individual stage derivatives, which is used inside the for loop on the next four lines. This loop forms the core of the method, and is essentially just (2.8) implemented in Python code, split over several lines for improved readability. The residual is returned as a single array of length `self.stages * self.neq`, as expected by the SciPy `root` function.
- Finally, the `advance` method calls the `solve_stages` to compute all the stage derivatives, and then advances the solution using a general implementation of (2.9).

With the general base class at hand, we can easily implement new solvers, simply by writing the constructors that define the method coefficients. The following code implements the implicit midpoint and the two- and three-stage Radau methods:

```
class ImplicitMidpoint(ImplicitRK):
    def __init__(self, f):
        super().__init__(f)
        self.stages = 1
        self.a = np.array([[1 / 2]])
        self.c = np.array([1 / 2])
        self.b = np.array([1])
```

```
class Radau2(ImplicitRK):
    def __init__(self, f):
        super().__init__(f)
        self.stages = 2
        self.a = np.array([[5 / 12, -1 / 12], [3 / 4, 1 / 4]])
        self.c = np.array([1 / 3, 1])
        self.b = np.array([3 / 4, 1 / 4])


class Radau3(ImplicitRK):
    def __init__(self, f):
        super().__init__(f)
        self.stages = 3
        sq6 = np.sqrt(6)
        self.a = np.array([[(88 - 7 * sq6) / 360,
                            (296 - 169 * sq6) / 1800,
                            (-2 + 3 * sq6) / (225)],
                           [(296 + 169 * sq6) / 1800,
                            (88 + 7 * sq6) / 360,
                            (-2 - 3 * sq6) / (225)],
                           [(16 - sq6) / 36, (16 + sq6) / 36, 1 / 9]])
        self.c = np.array([(4 - sq6) / 10, (4 + sq6) / 10, 1])
        self.b = np.array([(16 - sq6) / 36, (16 + sq6) / 36, 1 / 9])
```

Notice that we always define the method coefficients as NumPy arrays, even for the implicit midpoint method where they all contain a single number. This definition is necessary for the generic methods of the `ImplicitRK` class to work.

3.5.2 Base classes for SDIRK and ESDIRK methods

We could, in principle, implement both the SDIRK and ESDIRK methods in the same manner as the FIRK methods above, simply by defining the method coefficients in the constructor. The generic methods from the `ImplicitRK` base class will work fine even if we have $a_{ij} = 0$ for $j > i$. However, the motivation for deriving diagonally implicit methods is precisely to avoid solving these large systems of non-linear equations, so it does not make much sense to implement them in this way. Instead, we should utilize the structure of the method coefficients and solve for the stage variables sequentially. This requires rewriting the two methods `solve_stages` and `stage_eq` from the base class above. Once the stage derivatives have been computed, advancing the solution to the next step occurs in the same way for all RK methods, so the `advance` method can be left unchanged.

Considering first the SDIRK methods, we can implement these as subclasses of the `ImplicitRK` class, which enables some (moderate) code reuse and reflects the fact that SDIRK methods are indeed special cases of implicit

RK methods. Using the two-stage SDIRK method defined by (3.15) as an example, we get a better view of the tasks involved in the SDIRK methods if we write out the equations for the stage derivatives. Inserting the coefficients from (3.15) into (2.8)-(2.9) gives

$$k_1 = f(t_n + \gamma\Delta t, u_n + \Delta t\gamma k_1), \tag{3.17}$$

$$k_2 = f(t_n + \Delta t, u_n + \Delta t((1-\gamma)k_1 + \gamma k_2)), \tag{3.18}$$

$$u_{n+1} = u_n + \Delta t((1-\gamma)k_1 + \gamma k_2). \tag{3.19}$$

Here, (3.17) is nearly identical to the equation defining the stage derivative in the backward Euler method, the only difference being the factor γ in front of the arguments inside the function call. Furthermore, the only difference between (3.17) and (3.18) is the additional term $\Delta t(1-\gamma)k_1$ inside the function call. In general, any stage equation for any DIRK method can be written as

$$k_i = f(t_n + c_i\Delta t, u_n + \Delta t(\sum_{j=0}^{i-1} a_{ij}k_j + \gamma k_i)), \tag{3.20}$$

where the sum inside the function call only includes previously computed stages.

Given the similarity of (3.20) with the stage equation from the backward Euler method, it is natural to implement the SDIRK stage equation as a generalization of the `stage_eq` method from the `BackwardEuler` class. It is also convenient to place this method in an SDIRK base class, from which we may derive all specific SDIRK solver classes. Furthermore, since the stage equations can be written on this general form, it is not difficult to generalize the algorithm for looping through the stages and computing the individual stage derivatives. The base class can, therefore, contain general SDIRK versions of both the `stage_eq` and `solve_stages`, and the only task left in individual solver classes is to define the number of stages and the method coefficients. The complete base class implementation may look as follows.

```
class SDIRK(ImplicitRK):
    def stage_eq(self,k,c_i, k_sum):
        u, f, n, t = self.u, self.f, self.n, self.t
        dt = self.dt
        gamma = self.gamma

        return k - f(t[n]+c_i*dt,u[n]+dt*(k_sum+gamma*k))

    def solve_stages(self):
        u, f, n, t = self.u, self.f, self.n, self.t
        a, c = self.a, self.c
        s = self.stages

        k = f(t[n],u[n]) #initial guess for first stage
        k_sum = np.zeros_like(k)
        k_all = []
```

```
        for i in range(s):
            k_sum = sum(a_*k_ for a_,k_ in zip(a[i,:i],k_all))
            k = root(self.stage_eq,k,args=(c[i],k_sum)).x
            k_all.append(k)

        return k_all
```

The modified `stage_eq` method takes two additional parameters; the coefficient `c_i` corresponding to the current stage, and the array `k_sum` which holds the sum $\sum_{j=1}^{i-1} a_{ij}k_j$. These arguments need to be initialized correctly for each stage, and passed as additional arguments to the SciPy `root` function. For convenience, we also assume that the method parameter γ has been stored as a separate class attribute. With the `stage_eq` method implemented in this general way, the `solve_stages` method simply needs to update the weighted sum of previous stages (`k_sum`), and pass this and the correct c value as additional arguments to the SciPy `root` function. The implementation above implements this in a for loop which computes the stage derivatives sequentially and returns them as a list `k_all`.

As for the FIRK method classes, the only method we now need to implement specifically for each solver class is the constructor, in which we define the number of stages and the method coefficients. A class implementation of the method in (3.15) may look as follows.

```
class SDIRK2(SDIRK):
    def __init__(self,f):
        super().__init__(f)
        self.stages = 2
        gamma = (2-np.sqrt(2))/2
        self.gamma = gamma
        self.a = np.array([[gamma,0],
                           [1-gamma, gamma]])
        self.c = np.array([gamma,1])
        self.b = np.array([1-gamma, gamma])
```

Shifting our attention to the ESDIRK methods, these are identical to the SDIRK methods except for the first stage, and the potential for code reuse is obvious. Examining the two methods of the SDIRK base class above, we quickly conclude that the `stage_eq` method can be reused in an ESDIRK solver class, since the equations to be solved for each stage are identical for SDIRK and ESDIRK solvers. However, the `solve_stages` method needs to be modified, since there is no need to solve a non-linear equation for `k1`. The modifications can, however, be very small, since all stages $i > 1$ are identical. A possible implementation of the ESDIRK class can look as follows:

```
class ESDIRK(SDIRK):
    def solve_stages(self):
        u, f, n, t = self.u, self.f, self.n, self.t
        a, c = self.a, self.c
        s = self.stages
```

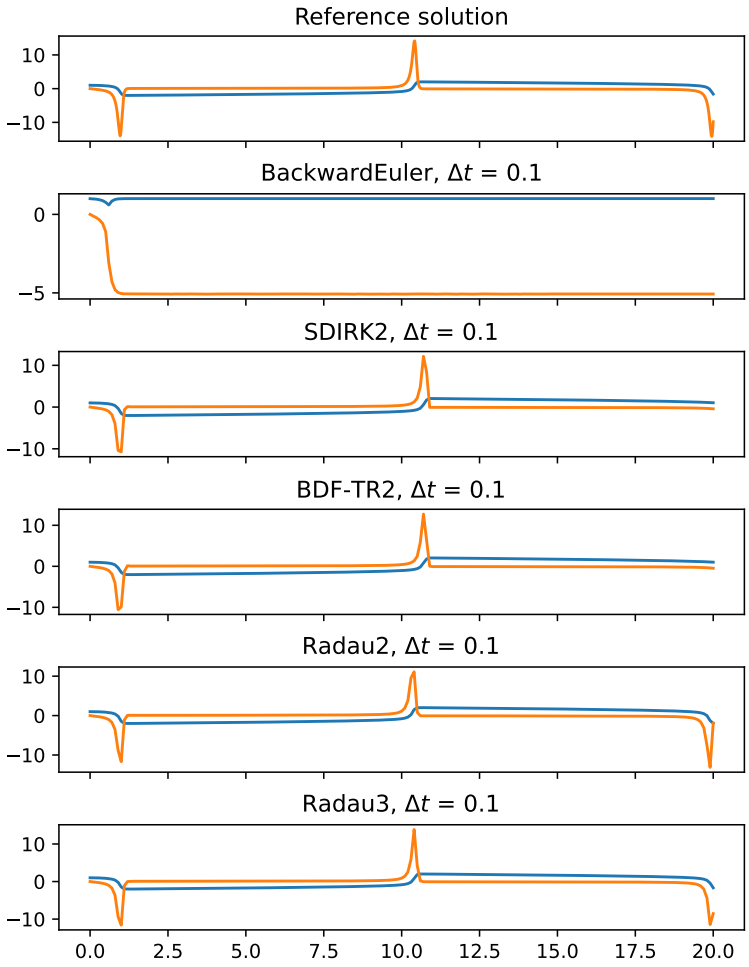

Fig. 3.6 Solutions of the Van der Pol model for $\mu = 10$ and $\Delta t = 0.1$, using implicit RK solvers of different accuracy.

```
        k = f(t[n],u[n])
        k_sum = np.zeros_like(k)
        k_all = [k]
        for i in range(1,s):
            k_sum = sum(a_*k_ for a_,k_ in zip(a[i,:i],k_all))
```

```
            k = root(self.stage_eq,k,args=(c[i],k_sum)).x
            k_all.append(k)

        return k_all
```

Comparing with the SDIRK base class above, the two methods look identical at first, but there are two small but important differences. The first is that the result of the first function evaluation `k = f(t[n],u[n])` is now used directly as the first stage, by setting `k_all = [k]`, instead of just serving as an initial guess for the nonlinear equation solver. The second is that the for-loop for computing the remaining stages starts at `i=1` rather than `i=0`.

With the ESDIRK base class at hand, we can implement individual ESDIRK methods simply by defining the constructor, for instance

```
class BDF_TR2(ESDIRK):
    def __init__(self,f):
        super().__init__(f)
        self.stages = 3
        gamma = 1-np.sqrt(2)/2
        beta = np.sqrt(2)/4
        self.gamma = gamma
        self.a = np.array([[0,0,0],
                           [gamma, gamma,0],
                           [beta,beta,gamma]])
        self.c = np.array([0,2*gamma,1])
        self.b = np.array([beta,beta,gamma])
```

Notice that these class implementations have some potential weaknesses. An obvious one is that the `solve_stages` methods of the SDIRK and ESDIRK classes are nearly identical, and most of the code is duplicated. Part of the purpose of implementing the solvers in a class hierarchy is to avoid code duplication, so this is obviously not optimal. However, avoiding duplicated code completely would in this case require refactoring the classes a bit, to split the tasks performed in `solve_stages` into several methods. Since these tasks belong quite naturally together, splitting them up could make the code more difficult to read and understand, and would also potentially make the code less computationally efficient. The latter should always be a consideration when implementing numerical methods, although it is not a strong focus of the present text.

Another choice that can be questioned in the `ESDIRK` class is that we retain the dimensions of the `self.a` coefficient array, and simply set the entire first row to zero. Storing these zeros is obviously not needed, and we could have omitted them and adjusted the for-loop in `solve_stages` accordingly. However, this choice would make the link between the code and the mathematical formulation of RK methods less obvious, and the benefits would be minimal.

Figure 3.6 illustrates the difference in accuracy between a number of IRK solvers. The chosen time step $\Delta t = 0.1$ is obviously too large for the backward Euler method, and the solution is not even close to the reference solution.

The other solvers are the three-stage SDIRK method of order two, the two-stage Radau method of order three, and three-stage Radau method of order five. We will see more examples of SDIRK methods in Chapter 4, when we introduce RK methods with adaptive time step.

Chapter 4
Adaptive time step methods

In practical computations, one seeks to achieve a desired accuracy with the minimum computational effort. For a given method, this requires finding the largest possible value of the time step Δt. In the previous chapters we always kept the step size constant through the solution interval, but this is rarely the most efficient approach, since the error depends on the characteristics of the solution in addition to the step size. In regions where the solution is smooth, large time steps can be used without introducing significant error, and in regions where the solution has rapid variations, a smaller time step must be employed. In this chapter we will extend the Runge-Kutta methods from the previous chapters, to methods that select the time step automatically to control the error in the solution.

4.1 A motivating example

Many ODE models of dynamic systems have solutions that vary rapidly in some intervals and are nearly constant in others. As a motivating example, we may consider a particular class of ODE models that which describe the so-called *action potential* of excitable cells. These models, first introduced by Hodgkin and Huxley [10], are important tools for studying the electrophysiology of cells such as neurons and different types of muscle cells. The main variable of interest is usually the transmembrane potential, which is the difference in electrical potential between the internals of a cell and its surroundings. When an excitable cell such as a neuron or a muscle cell is stimulated electrically, it triggers a cascade of processes in the cell membrane, leading to various ion channels opening and closing, and the membrane potential going from its resting negative state to approximately zero or slightly positive, before returning to the resting value. This process of *depolarization* followed by *repolarization* is called the action potential, and is illustrated in Figure 4.1. See, for instance, [11], for a comprehensive overview of the

Hodgkin-Huxley model and action potential models in general. The potential utility of adaptive time step methods is obvious from Figure 4.1. The solution changes rapidly during the action potential, but is approximately constant for long time intervals when the cell is at rest. Such behavior can be observed in many types of ODE models, and motivates methods that can adjust the time step to the properties of the solution. Commonly referred to as adaptive methods or methods with automatic time step control, these techniques are important parts of all modern ODE software.

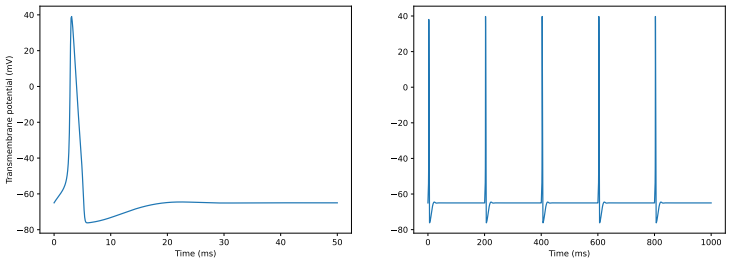

Fig. 4.1 Solution of the Hodgkin-Huxley model. The left panel shows a single action potential, while the right panel shows the result of stimulating the cell multiple times with a fixed period.

There are many possible approaches for selecting the time step automatically. One intuitive approach is to base the time step estimate on the dynamics of the solution, and select a small time step whenever rapid variations occur. This approach is commonly applied in adaptive solvers for partial differential equations (PDEs), where both the time step and space step can be chosen adaptively. It has also been successfully applied in specialized solvers for the action potential models mentioned above, see, e.g., [15], where the time step is simply selected based on the variations of the transmembrane voltage. However, this method may not be universally applicable and the criteria for choosing the time step must be carefully selected based on the characteristics of the problem at hand.

4.2 Choosing the time step based on the local error

The goal of an adaptive time stepping method is to control the error in the solution, and it is natural to base the step selection on some form of error estimate. In Section 1.5 we computed the error at the end of the solution interval, and used it to confirm the theoretical convergence of the method.

Such a global error could, in principle, also be useful for selecting the time step, since we could can simply redo the calculation with a smaller time step if the error is too large. However, for interesting ODE problems the analytical solution is not available, which complicates this form of error estimate. Furthermore, the goal for the adaptive time step methods is to select the time step dynamically as the solution progresses, to ensure that the final solution satisfies a given error tolerance. This goal requires a different approach, which is based on estimates of the local error for each step rather than the global error.

Assuming that we are able to compute an estimate for the local error for a given step, e_n, the goal is to choose the time step Δt_n so that the inequality

$$e_n < tol \tag{4.1}$$

is satisfied for all steps. There are two essential parts to the process of choosing Δt_n so that (4.1) is always satisfied. The first is that we always check the inequality after a step is performed. If it is satisfied we accept the step and proceed with step $n+1$ as normal, and if it is not satisfied we reject the step and try again with a smaller Δt_n. The second part of the procedure is to choose the next time step, that is, either Δt_{n+1} if the current step was accepted, or a new guess for Δt_n if it was rejected. We shall see that the same formula, derived from what we know about the local error, can be applied in both cases.

We first assume, for simplicity of notation, that step n was accepted with a time step Δt and a local error estimate $e_n < tol$. Our aim is now to choose Δt_{n+1} so that (4.1) is satisfied as sharply as possible to avoid wasting computations, so we want to choose Δt_{n+1} so that $e_{n+1} \lessapprox tol$. Recall from 1.5 that for a method of global order p, the local error is of order $p+1$, so we have

$$e_n \approx C(\Delta t_n)^{p+1} \tag{4.2}$$

$$e_{n+1} \approx C(\Delta t_{n+1})^{p+1} \tag{4.3}$$

where we have assumed that the error constant C is constant from one step to the next. Eq. (4.2) gives

$$C = \frac{e_n}{(\Delta t_n)^{p+1}},$$

and inserting into (4.3) gives

$$e_{n+1} \approx \frac{e_n}{(\Delta t_n)^{p+1}} (\Delta t_{n+1})^{p+1}.$$

We want to have $e_{n+1} \approx tol$, so we set

$$tol = e_{n+1} = \frac{e_n}{\Delta t_n^{p+1}} \Delta t_{n+1}^{p+1}$$

and rearrange to get the standard formula for time step selection

$$\Delta t_{n+1} = \left(\frac{tol}{e_n} \Delta t_n^{p+1} \right)^{1/(p+1)}.$$

We see that if $e_n \ll tol$ this formula will select a larger step size for the next step, while if $e_n \approx tol$ we get $\Delta t_{n+1} \approx \Delta t_n$. In practice, the formula is usually modified with a safety factor, i.e., we set

$$\Delta t_{n+1} = \eta \left(\frac{tol}{e_n} \Delta t_n^{p+1} \right)^{1/(p+1)}. \tag{4.4}$$

for some $\eta < 1$. The exact same formula can be used to choose a new step size Δt_n if the step was rejected, i.e., if $e_n > tol$.

While (4.4) gives a simple formula for the step size, and we shall see later that it works well for our example problems, more sophisticated methods have been derived. The problem of choosing the time step to control the error is an optimal control problem, and successful methods have been derived based on control theory, in order to control the error while avoiding rapid variations in the step size. See, for instance, [9] for details and examples of such methods.

4.3 Estimating the local error

The inequality (4.1) and formula (4.4) gives the necessary tools to select the time step based on the local error e_n. The remaining task is to come up with a method to estimate this error. It is, of course, not possible to compute it directly since the analytical solution is not available, but it can instead be estimated based on two numerical solution of different accuracy. The idea is simply to advance the solution from t_{n-1} to t_n twice, using two methods of different accuracy, giving us our regular solution u_n and a more accurate one $\hat{u}_n$. The difference $|\hat{u}_n - u_n|$ can then be used to estimate the local error for the solution u_n. The more accurate solution $\hat{u}_n$ can be computed in two ways; either by taking several "internal" time steps to advance from t_n to t_{n+1}, or by using a method with higher order of accuracy. The first approach is the foundation for a technique referred to as *step doubling*, where the solution $\hat{u}_{n+1}$ is computed with the same method used for u_{n+1}, but using two steps of length $\Delta t/2$ instead of one step Δt. This obviously makes $\hat{u}_{n+1}$ more accurate than u_{n+1}, but the difference is not very large, so the difference $|\hat{u}_{n+1} - u_{n+1}|$ cannot be used directly as an error estimate. However, an error estimate may be derived by combining this difference with the known order of the method, see [1] for details. The step doubling method is completely general and can

be used to provide a local error estimate for all ODE solvers. However, it is also computationally expensive, and most modern ODE software are based on other techniques. The second approach for computing $\hat{u}_n$, to use a method with higher order of accuracy, turns out to be particularly attractive for RK methods. We shall see in the next section that it is possible to construct so-called *embedded methods*, which provides an error estimate with very little additional computation.

4.3.1 Error estimates from embedded methods

For a numerical method of order p, a solution computed with a method of higher order for instance $p+1$, can be used to estimate the local error. Since Δt is small, we have $\Delta t^{p+1} \ll \Delta t^p$, and the error can be estimated directly as $e_n = |u_n - \hat{u}_n|$. It would be very expensive to compute these two solutions using two entirely different methods, but the error estimate can often be obtained more efficiently by embedded methods. An embedded method is a variation of a given RK method that uses the same stage computations as the original method, but achieves a different order of accuracy. Since most of the computational work in RK methods occurs in the stage computations, error estimates based on embedded methods are relatively cheap to evaluate.

For the general RK method defined by (2.8)-(2.9), an embedded method can be introduced by defining a separate set of weights $\hat{b}_i$, which advance the solution using the same k_i as the main method:

$$k_i = f(t_n + c_i\Delta t, y_n + \Delta t \sum_{j=1}^{s} a_{ij}k_j) \text{ for } i = 1, \ldots, s \tag{4.5}$$

$$u_{n+1} = u_n + \Delta t \sum_{i=1}^{s} b_i k_i, \tag{4.6}$$

$$\hat{u}_{n+1} = u_n + \Delta t \sum_{i=1}^{s} \hat{b}_i k_i. \tag{4.7}$$

Although the main idea is to reuse the same stage computations to compute both $\hat{u}_{n+1}$ and u_{n+1}, it is not uncommon to introduce one additional stage in the method to obtain the error estimate. An RK method with an embedded method for error estimation is often referred to as an RK pair of order $n(m)$, where n is the order of the main method and m the order of the method used for error estimation. Butcher tableaus for RK pairs are written exactly as before, but with one extra line for the additional coefficients $\hat{b}$:

$$\begin{array}{c|ccc} c_i & a_{11} & \cdots & a_{1s} \\ \vdots & \vdots & & \vdots \\ c_s & a_{s1} & \cdots & a_{ss} \\ \hline & b_1 & \cdots & b_s \\ & \hat{b}_1 & \cdots & \hat{b}_s \end{array}.$$

As an example we may consider the simplest possible embedded RK pair, which is obtained by combining Heun's method with the forward Euler method. The method is defined by the Butcher Tableau

$$\begin{array}{c|cc} 0 & 0 & \\ 1 & 1 & \\ \hline & 1 & 0 \\ \hline & 1/2 & 1/2 \end{array}, \tag{4.8}$$

which translates to the following formulas for advancing the two solutions:

$$\begin{aligned} k_1 &= f(t_n, u_n), \\ k_2 &= f(t_n + \Delta t, u_n + \Delta t k_1), \\ u_{n+1} &= u_n + \Delta t k_1, \\ \hat{u}_{n+1} &= u_n + \Delta t/2(k_1 + k_2). \end{aligned}$$

In the next section we will see how this method pair can be implemented as an extension of the `ODESolver` hierarchy introduced earlier, before we introduce more advanced embedded RK methods in Section 4.5.

4.4 Implementing an adaptive solver

In the previous chapters we have been able to reuse significant parts of the original `ODESolver` base class for all the RK method. The subclasses for the explicit RK methods needed to reimplement the `advance` method, while the implicit methods required a few additional methods, and it was also convenient to redesign the classes to define the method coefficients as attributes in the constructor. However, all the subclasses could reuse the `solve` method which contained the main solver loop. A quick inspection of this method reveals that the assumption of a fixed number of time steps is quite fundamental to the implementation, since it is based on a for-loop and NumPy arrays with fixed size. With an adaptive step size the number of steps is obviously not fixed, and we therefore need to change the `solve` method significantly. In fact, the only part of the original `ODESolver` class that can be reused directly is the `set_initial_condition`, which is obviously a very moderate benefit. However, it can still make sense to implement the adaptive methods as sub-

classes of `ODESolver`, to benefit from this tiny code reuse and to highlight that an adaptive solver is in fact a special case of a general ODE solver. Since most of the new functionality needed by adaptive solvers is generic to all adaptive solvers, it makes sense to implement them in a general base class. In summary, the following changes and additions are needed:

- A complete rewrite of the `solve` method, to replace the for-loop and NumPy arrays with lists and a while loop. Lists are usually not preferred for computational tasks, but for adaptive time step methods their flexible size makes them attractive. It is also natural to add more parameters to the `solve` function, to let the user specify the tolerance and a maximum and minimum step size.
- The `advance` method needs to be updated to return both the updated solution and the error estimate.
- The step selection formula in (4.4) must be implemented in a separate method.
- Adaptive methods usually include a number of additional parameters, such as the safety factor η and the order p used in (4.4). These parameters are conveniently defined as attributes in the constructor.

An implementation of the adaptive base class may look as follows:

```
from ODESolver import *
from math import isnan, isinf

class AdaptiveODESolver(ODESolver):
    def __init__(self, f, eta=0.9):
        super().__init__(f)
        self.eta = eta

    def new_step_size(self,dt,loc_error):
        eta = self.eta
        tol = self.tol
        p = self.order
        if isnan(loc_error) or isinf(loc_error):
            return self.min_dt

        new_dt = eta * (tol/loc_error)**(1/(p+1)) * dt
        new_dt = max(new_dt,self.min_dt)
        return min(new_dt,self.max_dt)


    def solve(self,t_span,tol=1e-3,max_dt=np.inf,min_dt=1e-5):
        """Compute solution for t_span[0] <= t <= t_span[1]"""
        t0,T = t_span
        self.tol = tol
        self.min_dt = min_dt
        self.max_dt = max_dt
        self.t = [t0]

        if self.neq == 1:
```

```
            self.u = [np.asarray(self.u0).reshape(1)]
        else:
            self.u = [self.u0]

        self.n = 0
        self.dt = 0.1/np.linalg.norm(self.f(t0,self.u0))

        loc_t = t0
        while loc_t < T:
            u_new, loc_error = self.advance()
            if loc_error < tol or self.dt < self.min_dt:
                loc_t += self.dt
                self.t.append(loc_t)
                self.u.append(u_new)
                self.dt = self.new_step_size(self.dt,loc_error)
                self.dt = min(self.dt, T-loc_t, max_dt)
                self.n += 1
            else:
                self.dt = self.new_step_size(self.dt,loc_error)
        return np.array(self.t), np.array(self.u)
```

The constructor should be self-explanatory, but the other two methods deserve a few comments. The `step_size` method is essentially a Python implementation of (4.4), with tests to ensure that the selected step size is within the user defined range. We have also added a check which ensures that if the computed error is infinity or not a number (`inf` or `nan`) the new step size is automatically set to the minimum step size. This test is important for the robustness of the solver, since explicit methods will often diverge and return `inf` or `nan` values if applied to very stiff problems. Checking for these values and setting a low step size if they occur will therefore reduce the risk of complete solver failure. The small step size will still make the computation inefficient, but this is far better than unexpected failure. The `solve` method has also been substantially changed from the `ODESolver` version. First, the parameter list has been expanded to include the tolerance as well as the maximum and minimum time step. These are all stored as attributes and used in the main loop. The truly significant changes start with the initialization of the attributes `self.t` and `self.u`, which are now lists of length one rather than fixed size NumPy arrays. Notice also the somewhat cumbersome initialization of `self.u`, which includes an if-test that checks if we solve a scalar ODE or a system. This initialization ensures that for scalar equations, `self.u[0]` is a one-dimensional array of length one, rather than a zero-dimensional array. The actual contents of these two data structures is the same, i.e., a single number, but they are treated differently by some NumPy tools and it is useful to make sure that `self.u[0]`,`self.u[1]`, and so forth all have the same dimensions. The first step size is then calculated using a simplified version of the algorithm outlined in [8]. The for-loop has been replaced by a while-loop, since the number of steps is initially unknown. The call to the `advance`-method gives the updated solution and the estimated local error, and we proceed to check if the local error is lower than the tolerance. If it is,

the new time point and solution are appended to the corresponding lists, and the next time step is chosen based on the current one and the local error. The min and max operations are included to ensure that the time step is within the selected bounds, and that the simulation actually ends at the final time `T`. If the constraint `loc_error < tol` is not satisfied, we simply compute a new time step and try again, without updating the lists for the time and the solution.

While the `solve` loop in the `AdaptiveODESolver` class is obviously a lot more complex than the earlier versions, it should be noted that it is still a very simple version of an adaptive solver. The aim here is to present the fundamental ideas and promote the general understanding of how these solvers are implemented, and we therefore only include the most essential parts. Important limitations and simplifications include the following:

- As noted above, the step size selection in (4.4), implemented in `step_size`, could be replaced with more sophisticated versions. See, for instance, [3,9] for details.
- The formula for selecting the initial step is very simple, and is mainly suitable for avoiding extremely bad choices for the initial step size. More sophisticated algorithms have been derived, and we refer to, for instance, [8,9] for details.
- The first `if`-test inside the solver loop is not the most robust, since it will accept the stem and move forward if the minimum step size is reached, even if the error is too large. A robust solver should in this case give the user a warning that the requested tolerance cannot be reached.

In spite of these and other limitations, the adaptive solver class works as intended, and captures the essential behavior of adaptive ODE solvers.

With the `AdaptiveODESolver` base class at hand, subclasses for specific solvers can be implemented by writing specific versions of the `advance` method and the constructor, since the order of the method is used in the time step selection and therefore needs to be defined as an attribute. For the Euler-Heun method pair listed above, a suitable implementation may look as follows:

```
class EulerHeun(AdaptiveODESolver):
    def __init__(self, f, eta=0.9):
        super().__init__(f,eta)
        self.order = 1

    def advance(self):
        u, f, t = self.u, self.f, self.t
        dt = self.dt
        k1 = f(t[-1], u[-1])
        k2 = f(t[-1] + dt, u[-1] + dt*k1)
        high = dt/2*(k1+k2)
        low = dt*k1

        unew = u[-1] + low
```

```
        error = np.linalg.norm(high-low)
        return unew, error
```

After computing the two stage derivatives `k1` and `k2`, the method computes the high and low order solution updates. The low order is used to advance the solution, while the difference between the two provides the error estimate. The method returns the updated solution and the error, as needed by the `solve` method implemented in the base class above.

Since we have two methods with different accuracy, we may ask whether it would be better to advance the solution using the most accurate rather than the least accurate method. This choice will, of course, give a reduced local error, but the obvious downside is that we would no longer have a proper error estimate. We can use the more accurate solution to estimate the error of the less accurate, but not the other way around. However, the approach, known as *local extrapolation* [8] is still used by many popular RK pairs, as we shall see in examples below. Even if the error estimate is then no longer a proper error estimate for the method used to integrate the solution, it still works well as a tool for selecting the time step. In the implementation above it is very easy to play around with this choice, by replacing `low` with `high` in the assignment of `unew`, and check the effect on the error and the number of time steps.

4.5 More advanced embedded RK methods

There are numerous examples of explicit RK pairs of higher order than the 1(2) pair defined by (4.8). We will not provide an exhaustive list here, but mention two particularly popular methods, which have been implemented in various software packages. The first is a method by Fehlberg, often referred to as the Fehlberg 4(5) or simply the RKF45 method [5]. The Butcher tableau is

$$
\begin{array}{c|ccccccc}
0 & & & & & & & \\
\frac{1}{4} & \frac{1}{4} & & & & & & \\
\frac{3}{8} & \frac{3}{32} & \frac{9}{32} & & & & & \\
\frac{12}{13} & \frac{1932}{2197} & -\frac{7200}{2197} & \frac{7296}{2197} & & & & \\
1 & \frac{439}{216} & -8 & \frac{3680}{513} & -\frac{845}{4104} & & & \\
\frac{1}{2} & -\frac{8}{27} & 2 & -\frac{3544}{2565} & \frac{1859}{4104} & -\frac{11}{40} & & \\
\hline
 & \frac{25}{216} & 0 & \frac{1408}{2565} & \frac{2197}{4104} & -\frac{1}{5} & 0 & \\
 & \frac{16}{135} & 0 & \frac{6656}{12825} & \frac{28561}{56430} & -\frac{9}{50} & \frac{2}{55} &
\end{array}
\quad , \tag{4.9}
$$

Here, the first line of b-coefficients (b_i) yields a fourth order method, while the bottom line ($\hat{b}_i$) gives a method of order five. The implementation of the RKF45 method is similar to the Euler-Heun pair, although the number of stages and coefficients makes the `advance` method considerably more complex:

```
class RKF45(AdaptiveODESolver):
    def __init__(self, f, eta=0.9):
        super().__init__(f, eta)
        self.order = 4

    def advance(self):
        u, f, t = self.u, self.f, self.t
        dt = self.dt
        c2 = 1/4; a21 = 1/4;
        c3 = 3/8; a31 = 3/32; a32 = 9/32
        c4 = 12/13; a41 = 1932/2197; a42 = -7200/2197; a43 = 7296/2197
        c5 = 1; a51 = 439/216; a52 = -8; a53 = 3680/513; a54 = -845/4104
        c6 = 1/2; a61 = -8/27; a62 = 2; a63 = -3544/2565;
        a64 = 1859/4104; a65 = -11/40
        b1 = 25/216; b2 = 0; b3 = 1408/2565; b4 = 2197/4104;
        b5 = -1/5; b6 = 0
        bh1 = 16/135; bh2 = 0; bh3 = 6656/12825; bh4 = 28561/56430;
        bh5 = -9/50; bh6 = 2/55

        k1 = f(t[-1], u[-1])
        k2 = f(t[-1] + c2*dt, u[-1] + dt*(a21*k1))
        k3 = f(t[-1] + c3*dt, u[-1] + dt*(a31*k1+a32*k2))
        k4 = f(t[-1] + c4*dt, u[-1] + dt*(a41*k1+a42*k2+a43*k3))
        k5 = f(t[-1] + c5*dt, u[-1] + dt*(a51*k1+a52*k2+a53*k3+a54*k4))
        k6 = f(t[-1] + c6*dt, u[-1] +
               dt*(a61*k1+a62*k2+a63*k3+a64*k4+a65*k5))

        low  = dt*(b1*k1+b3*k3+b4*k4+b5*k5)
        high = dt*(bh1*k1+bh3*k3+bh4*k4+bh5*k5+bh6*k6)

        unew = u[-1] + low
        error = np.linalg.norm(high-low)

        return unew, error
```

The `advance` method could obviously be written more compactly, but we chose to keep the structure of the explicit RK methods introduced earlier.

Another famous and widely used pair of ERK methods is the Dormand-Prince method [4], which is a seven-stage method with the following coefficients:

0							
$\frac{1}{5}$	$\frac{1}{5}$						
$\frac{3}{10}$	$\frac{3}{40}$	$\frac{9}{40}$					
$\frac{4}{5}$	$\frac{44}{45}$	$-\frac{56}{15}$	$\frac{32}{9}$				
$\frac{8}{9}$	$\frac{19372}{6561}$	$-\frac{25360}{2187}$	$\frac{64448}{6561}$	$-\frac{212}{729}$			
1	$\frac{9017}{3168}$	$-\frac{355}{33}$	$\frac{46732}{5247}$	$\frac{49}{176}$	$-\frac{5103}{18656}$		
1	$\frac{35}{84}$	0	$\frac{500}{1113}$	$\frac{125}{192}$	$-\frac{2187}{6784}$	$\frac{11}{84}$	
y_n	$\frac{35}{384}$	0	$\frac{500}{1113}$	$\frac{125}{192}$	$-\frac{2187}{6784}$	$\frac{11}{84}$	0
$\hat{y}_n$	$\frac{5179}{57600}$	0	$\frac{7571}{16695}$	$\frac{393}{640}$	$-\frac{92097}{339200}$	$\frac{187}{2100}$	$\frac{1}{40}$

This method has been optimized for the local extrapolation approach mentioned above, where the highest order method is used to advance the solution and the less accurate method is only used for step size selection. The implementation is otherwise similar to the RKF45 method. The Dormand-Prince method has been implemented in numerous software tools, including the popular `ode45` function in Matlab (The Math Works, Inc. MATLAB. Version 2023a).

Implicit RK methods can also be equipped with embedded methods. The fundamental idea is exactly the same as for explicit methods, although the step size selection tends to be more challenging for stiff problems. The most obvious constraint is that for stiff problems, both the main method and the error estimator need to have good stability properties. Stiff problems are also known to be more challenging for the error control algorithms, and simple algorithms such as (4.4) often suffer from large fluctuations in step size and local error. We refer to, for instance, [1, 9] for a detailed discussion of these challenges, and here mainly present a selection of well known methods. For instance, the TR-BDF2 method in (3.16) can be extended to include a third order method for error estimation. The extended Butcher tableau is

$$\begin{array}{c|ccc} 0 & 0 & & \\ 2\gamma & \gamma & \gamma & 0 \\ 1 & \beta & \beta & \gamma \\ \hline & \beta & \beta & \gamma \\ & \frac{1-\beta}{3} & \frac{3\beta+1}{3} & \frac{\gamma}{3} \end{array}, \tag{4.10}$$

with $\gamma = 1 - \sqrt{2}/2$ and $\beta = \sqrt{2}/4$, and the bottom line of coefficients defines the third order method. This third order method is not L-stable, and for stiff problems it is therefore preferable to advance the solution using the second-order method and use the more accurate one for time step control. Ideally we would like both methods of an embedded RK pair to be L-stable, but this is often impossible to achieve and we need to accept somewhat weaker stability requirements for the error estimator, see, for instance, [13].

When implementing the adaptive TR-BDF2 and other implicit methods, we need to combine features of the `AdaptiveODESolver` class above with the tools from the `ImplicitRK` hierarchy introduced in Chapter 3. Specifically, an adaptive implicit RK methods needs the `solve` and `new_step_size` method from `AdaptiveODESolver`, while all the code for computing the stage derivatives can be reused directly from the `ImplicitRK` classes. A convenient way to reuse functionality from two different classes is to use *multiple inheritance*, where we define a new class a subclass of two different base classes. For instance, a base class for adaptive ESDIRK methods may look like

```
class AdaptiveESDIRK(AdaptiveODESolver,ESDIRK):
```

which simply states that the new class inherits all the methods from both the `AdaptiveODESolver` class and the `ImplicitRK` class. The general de-

sign of the `ImplicitRK` class above was to define the method coefficients in the constructor and use a generic `advance` method, and it is convenient to use the same method for the adaptive implicit methods. We then need to override the `advance` method from `ImplicitRK` in our `AdaptiveImplicitRK` base class, since we need the method to return the error in addition to the updated solution. All other methods can be reused directly from either `AdaptiveODESolver` or `ImplicitRK`, so a suitable implementation of the new class may look like

```
class AdaptiveESDIRK(AdaptiveODESolver,ESDIRK):
    def advance(self):
        b = self.b
        e = self.e
        u = self.u
        dt = self.dt
        k = self.solve_stages()
        u_step = dt*sum(b_*k_ for b_,k_ in zip(b,k))
        error = dt*sum(e_*k_ for e_,k_ in zip(e,k))
        u_new = u[-1] + u_step
        error_norm = np.linalg.norm(error)
        return u_new, error_norm
```

Here, we assume that the constructor defines all the RK method parameters used earlier, and in addition a set of parameters `self.e`, defined by $e_i = b_i - \hat{b}_i$, for $i = 1, \ldots, n$, which are used in the error calculations. Except for the two lines computing the error, the method is identical to the generic `advance` method from the `ImplicitRK` class, which was used by all the subclasses. Therefore, it may be natural to ask whether we should have put this method in a general base class for implicit RK methods, for instance named `AdaptiveImplicitRK`, and then it could be used in adaptive versions of both the SDIRK, ESDIRK, and Radau classes. However, adaptive versions of the Radau methods use a slightly different calculation of the error, since for a Radau method of order p it is not possible to construct an embedded method of order $p-1$. For the adaptive solvers the advance method is therefore slightly less general, and it is convenient to implement it separately for the ESDIRK methods. We will not present adaptive versions of the Radau methods here, but the details may be found in [9].

Although multiple inheritance provides a convenient way to reuse the functionality of our existing classes, it comes with the risk of somewhat complex and confusing class hierarchies. In particular, the fact that our `AdaptiveESDIRK` class inherits from `AdaptiveODESolver` and `ESDIRK`, which are both subclasses of `ODESolver`, may give rise to a well-known ambiguity referred to as the *diamond problem*. The problem would arise if, for instance, we were to define a method in `ODESolver`, override it with special versions in both `AdaptiveODESolver` and `ESDIRK`, and then call it from an instance of `AdaptiveESDIRK`. Would we then call the version implemented in `AdaptiveODESolver` or the one in `ESDIRK`? The answer is determined by Python's so-called *method resolution order* (MRO), which decides which

method to inherit first based on its "closeness" in the class hierarchy and then on the order of the base classes in the class definition. In our particular example the `AdaptiveESDIRK` class is equally close to `AdaptiveODESolver` and `ESDIRK`, since it is a direct subclass of both. The method called would therefore be the version from `AdaptiveODESolver`, since this is listed first in the class definition. In our relatively simple class hierarchy there are no such ambiguities, and even if we use multiple inheritance it should not be too challenging to determine which methods are called, but it is a potential source of confusion that is worth being aware of.

With the `AdaptiveESDIRK` base class available, an adaptive version of the TR-BDF2 method may be implemented as

```
class TR_BDF2_Adaptive(AdaptiveESDIRK):
    def __init__(self,f,eta=0.9):
        super().__init__(f,eta)
        self.stages = 3
        self.order = 2
        gamma = 1-np.sqrt(2)/2
        beta = np.sqrt(2)/4
        self.gamma = gamma
        self.a = np.array([[0,0,0],
                           [gamma, gamma,0],
                           [beta,beta,gamma]])
        self.c = np.array([0,2*gamma,1])
        self.b = np.array([beta,beta,gamma])
        bh = np.array([(1-beta)/3,(3*beta + 1)/3, gamma/3])
        self.e = self.b-bh
```

To illustrate the use of this solver class, we may return to the Hodgkin-Huxley model considered at the start of this chapter. Assuming that we have implemented the model as a class in a file `hodgkinhuxley.py`, the following code solves the model and plots the transmembrane potential:

```
from AdaptiveImplicitRK import TR_BDF2_Adaptive
from hodgkinhuxley import HodgkinHuxley
import matplotlib.pyplot as plt

model = HodgkinHuxley()
u0 = [-45,0.31,0.05,0.59]
t_span = (0,50)
tol = 0.01

solver = TR_BDF2_Adaptive(model)
solver.set_initial_condition(u0)

t,u = solver.solve(t_span,tol)

plt.plot(t,u[:,0])
plt.show()
```

A plot of the solution is shown in Figure 4.2. The +-marks show the time steps chosen by the adaptive TR-BDF2 solver, and it is easy to see that large

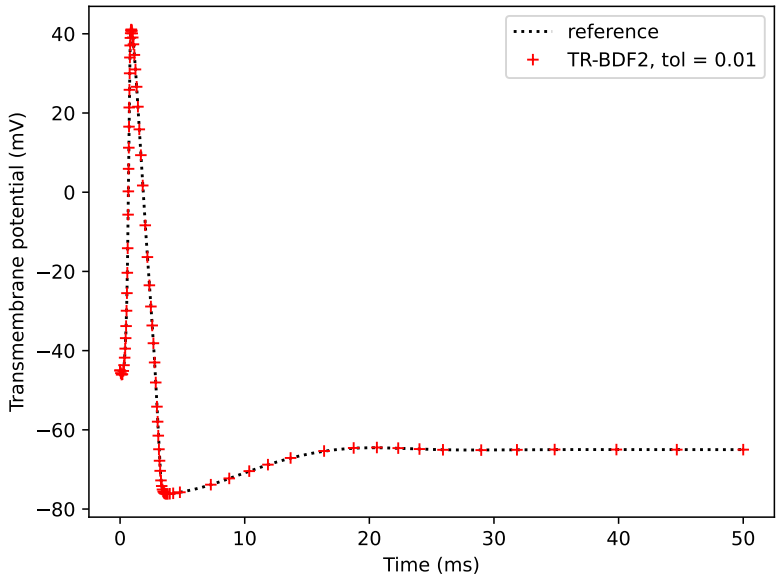

Fig. 4.2 Solution of the Hodgkin-Huxley model. The solid line is a reference solution computed with SciPy `solve_ivp`, while the +-marks are the time steps chosen by the adaptive TR-BDF2 solver.

time steps are used in quiescent regions while smaller steps are used where the solution varies rapidly. A more quantitative view of the solver behavior, for three different solvers, is shown in the table below. Each method has been applied with three different tolerance values, for the time interval from 0 to 50ms, and with default choices of the maximum and minimum time steps. The column marked "Error" is then an estimate of the global error, based on a reference solution computed with SciPy's `solve_ivp`, the "Steps" column is the number of accepted time steps, "Rejected" is the total number of rejected steps, and the two last columns are the minimum and maximum time steps that occurred during the computation.

Solver	Tolerance	Error	Steps	Rejected	Δt_{max}	Δt_{min}
TR-BDF2	1.000	0.0336961	24	9	10.533	0.00791
TR-BDF2	0.100	0.0175664	43	14	9.705	0.00791
TR-BDF2	0.010	0.0028838	83	22	5.328	0.00791
RKF45	1.000	0.6702536	192	113	2.204	$1.0 \cdot 10^{-5}$
RKF45	0.100	0.0934201	118	58	1.093	$1.0 \cdot 10^{-5}$
RKF45	0.010	0.0054336	123	34	1.297	0.00791
EulerHeun	1.000	0.7790353	158	35	1.849	0.00791
EulerHeun	0.100	0.0016577	220	40	0.836	0.00791
EulerHeun	0.010	0.0014654	432	36	0.918	0.00251

The numbers in this table illustrate a number of well-known properties and limitations of adaptive ODE solvers. First, we may observe that there is no close relationship between the selected tolerance and the resulting error. The error gets smaller when we reduce the tolerance, and for this particular case the error is always smaller than the specified tolerance, but the error varies substantially between the different methods. As described at the start of this chapter, the time step is selected to control the *local error*, and although we expect the global error to decrease as we reduce the tolerance, we have no guarantee that the global error is lower than the tolerance. A second observation we can make is that the RKF45 and EulerHeun methods both perform rather poorly, and show somewhat inconsistent behavior as the tolerance is reduced. For instance, the RKF45 method uses the largest number of steps, and also rejects the largest number of steps, for the highest tolerance. This behavior is caused by the fact that the Hodgkin-Huxley model is stiff, and the time step for the explicit methods is caused mainly by stability and not accuracy. The Δt_{min} of $1.0 \cdot 10^{-5}$ is caused by divergence issues which automatically sets the time step to the specified lower bound. For most of the other combinations of method and tolerance the smallest time step observed is the first one, selected by the simple formula inside the `solve` method. This could obviously be improved, and the general performance of RKF45 for stiff problems could be improved by a sophisticated step size controller. However, the performance will never be on the level of implicit solvers, since the time step will be dictated by stability rather than accuracy.

The ideas and tools introduced in this chapter are fundamental for all RK methods with error control and automatic time step selection. The main ideas are fairly simple, and, as illustrated in Figure 4.2, give rise to methods that effectively adapt the time step to control the error. There are, however, numerous matters to be considered regarding the practical implementation of the methods, and we have merely scratched the surface of these. As mentioned above, obvious points of improvement include the time step control formula in (4.4), for which more sophisticated models have been derived based on control theory. [7] The simple formula used for selecting the first time could also be improved, as indicated by the fact that the smallest step Δt_{min} is

the first one for most of the solvers in the table above. Furthermore, adjusted error estimates have been proposed, see [9], which are more suitable for stiff systems. For a more detailed and complete discussion of automatic time step control, we refer to [1] and [8, 9].

Chapter 5
Modeling infectious diseases

Throughout this book we have focused entirely on *solving* ODEs, and we have not spent much time considering the origin of the equations or what they may be used for. In the present chapter we focus on *modeling* with ODEs, by considering a very famous and widely used class of ODE models that describe the spread of infectious diseases. This group of models provides a good example of how a very complex phenomenon can be modeled with relatively simple systems of ODEs. We will derive the models from a set of fundamental assumptions, and discuss the limitations that result from these assumptions. Although we only consider one phenomenon and one class of models, the fundamental steps of the modeling process are quite generic, and can be applicable for a wide range of real-world phenomena.

5.1 Derivation of the SIR model

We want to model how infectious diseases spread in a population. This is a topic of obvious scientific and societal interest, which has been studied by scientists for centuries and, for obvious reasons, received enormous attention in recent years. The classical model for predicting the dynamics of an epidemic was derived by Kermack and McKendrick [12] in the early 1900s, and is referred to as the SIR model, since it describes the three categories Susceptible, Infected, and Recovered (or, alternatively, Removed). The spread of disease in a population is a very complex process, and in order to derive an ODE based model we need to make a number of simplifying assumptions. The most important one is that we do not consider individuals, just the total population and the number of people that move between the three categories. The population is assumed to be perfectly mixed in a confined area, which means that we do not consider spatial transport of the disease, just temporal evolution. The first model we will derive is very simple, but we shall see that it can easily be extended to models that are used world-wide by health

authorities to predict the spread of diseases such as Covid19, flu, ebola, HIV, etc.

In the first version of the model we will keep track of the three categories of people mentioned above:

- **S**: susceptibles - who can get the disease
- **I**: infected - who have developed the disease and can infect susceptibles
- **R**: recovered - who have recovered and become immune

We represent these as mathematical quantities $S(t)$, $I(t)$, $R(t)$, which represent the number of people in each category. The goal is now to derive a set of equations for $S(t)$, $I(t)$, $R(t)$, and then solve these equations to predict the spread of the disease.

To derive the model equations, we first consider the dynamics in a time interval Δt, and our goal is to derive mathematical expressions for how many people that move between the three categories in this time interval. The key part of the model is the description of how people move from S to I, i.e., how susceptible individuals get the infection from those already infected. Infectious diseases are (mainly) transferred by direct interactions between people, so we need mathematical descriptions of the number of interactions between susceptible and infected individuals. We make the following assumptions:

- An individual in the S category interacts with an approximately constant number of people each day, so the number of interactions in a time interval Δt is proportional to Δt.
- The probability of one of these interactions being with an infected person is proportional to the ratio of infected individuals to the total population, i.e., to I/N, with $N = S + I + R$.

Based on these assumptions, the probability that a single susceptible person gets infected is proportional to $\Delta t I/N$. The total number of infections can be written as $\beta SI/N$, for some constant β, which represents the probability that an infected person meets and infects a susceptible person. The value of β depends both on the infectiousness of the disease and the behavior of the population, as will be discussed in more detail below. The infection of new individuals represents a reduction in S and a corresponding gain in I, so we have

$$S(t+\Delta t) = S(t) - \Delta t \beta \frac{S(t)I(t)}{N},$$
$$I(t+\Delta t) = I(t) + \Delta t\, \beta \frac{S(t)I(t)}{N}.$$

These two equations represent the key component of all the models considered in this chapter. They are formulated as *difference equations*, and we will see below that they can easily be transformed to ODEs. More advanced models are typically derived by adding more categories and more transitions between

them, but the individual transitions are very similar to the ones presented here.

Fig. 5.1 Graphical representation of the simplest SIR-model, where people move from being susceptible (S) to being infected (I) and then reach the recovered (R) category with immunity against the disease.

We also need to model the transition of people from the I to the R category. Again considering a small time interval Δt, it is natural to assume that a fraction $\Delta t\,\nu$ of the infected recover and move to the R category. Here ν is a constant describing the time dynamics of the disease. The increase in R is given by

$$R(t+\Delta t) = R(t) + \Delta t\,\nu I(t),$$

and we also need to subtract the same term in the balance equation for I, since the people move from I to R. We get

$$I(t+\Delta t) = I(t) + \Delta t\,\beta S(t) I(t) - \Delta t\,\nu I(t).$$

We now have three equations for S, I, and R:

$$S(t+\Delta t) = S(t) - \Delta t\,\beta \frac{S(t)I(t)}{N} \tag{5.1}$$

$$I(t+\Delta t) = I(t) + \Delta t\,\beta \frac{S(t)I(t)}{N} - \Delta t \nu I(t) \tag{5.2}$$

$$R(t+\Delta t) = R(t) + \Delta t\,\nu I(t). \tag{5.3}$$

These equations are a system of difference equations, as discussed in more detail in Appendix A. We could easily solve the equations as such, using techniques from Appendix A, but it is even more convenient to formulate the model as a system of ODEs and apply the ODE solvers derived in previous chapters.

To turn the difference equations into ODEs, we first divide all equations by Δt and rearrange, to get

$$\frac{S(t+\Delta t) - S(t)}{\Delta t} = -\beta \frac{S(t)I(t)}{N}, \tag{5.4}$$

$$\frac{I(t+\Delta t) - I(t)}{\Delta t} = \beta t \frac{S(t)I(t)}{N} - \nu I(t), \tag{5.5}$$

$$\frac{R(t+\Delta t) - R(t)}{\Delta t} = \nu I(t). \tag{5.6}$$

We see that by letting $\Delta t \to 0$, we get derivatives on the left-hand side:

$$S'(t) = -\beta \frac{SI}{N}, \tag{5.7}$$

$$I'(t) = \beta \frac{SI}{N} - \nu I \tag{5.8}$$

$$R'(t) = \nu I, \tag{5.9}$$

where, as above, $N = S + I + R$.[1] If we add the three equations we see that $N'(t) = S'(t) + I'(t) + R'(t) = 0$, so the total population N is constant. The equations (5.7)-(5.9) is a system of three ODEs, which we will solve for the unknown functions $S(t)$, $I(t)$, $R(t)$. To solve the equations we need to specify initial conditions $S(0)$ (many), $I(0)$ (few), and $R(0)$ (=0), as well as the parameters β and ν.

For practical applications of the model, estimating the parameters is usually a major challenge. We can estimate ν from the fact that $1/\nu$ is the average recovery time for the disease, which is usually possible to determine from early cases. The infection rate β, on the other hand, lumps a lot of biological and sociological factors into a single number, and it is usually very difficult to estimate for a new disease. It depends both on the biology of the disease itself, essentially how infectious it is, and on the interactions of the population. In a global pandemic the behavior of the population varies between different countries, and it will typically change with time, so β must usually be adapted to different regions and different phases of the disease outbreak.

Epidemiologists often refer to the basic reproduction number $R0$ of an epidemic, which is the average number of new persons that an infected person infects. The critical number is $R0 = 1$, since if $R0 < 1$ the epidemic will decline, while for $R0 > 1$ it will grow exponentially. In the simple model considered here, the relationship between $R0$ and β is $R0 = \beta/\nu$, since β measures the number of disease transmissions per time, and $1/\nu$ is the mean duration of the infectious period. Be aware of the potential confusion between the R category of the SIR, in particular its initial value $R(0)$, and the basic reproduction number $R0$. These quantities are not directly related, and the notation is obviously not optimal, but we use it here since it is very established in the field.

Although the system (5.7)-(5.9) looks quite simple, analytical solutions cannot easily be derived. For particular applications it is common to make simplifications that allow simple analytical solutions. For instance, when studying the early phase of an epidemic one is mostly interested in the I

[1] A simpler version of the SIR model is also quite common, where the disease transmission term is not scaled with N. Eq. (5.8) then reads $S' = -\beta SI$, and (5.8) is modified similarly. Since N is constant the two models are equivalent, but the version in (5.7)-(5.9) is more common in real-world applications and gives a closer relation between β and common parameters such as the reproduction number.

category, and since the number of infected cases in this phase is low compared with the entire population we may assume that S is approximately constant and equal to N. Inserting $S \approx N$ turns (5.8) into a simple equation describing exponential growth, with solution

$$I(t) = I_0 e^{(\beta - \nu)}. \tag{5.10}$$

Such an approximate formula may be very useful, in particular for estimating the parameters of the model. In the early phase of an epidemic the number of infected people typically follows an exponential curve, and we can fit the parameters of the model so that (5.10) fits the observed dynamics. We can also relate the behavior of this simple model to the basic reproduction number $R0$ introduced above. We have $R0 = \beta/\nu$, and therefore $R0 > 1$ makes the exponent of (5.10) positive, while for $R0 < 1$ the exponent is negative and $I(t)$ declines. However, if we want to describe the full dynamics of the epidemic we need to solve the complete system of ODEs, and then we need numerical solvers like the ones developed in the previous chapters.

Solving the SIR model with the `ODESystem` class hierarchy. We can easily solve the SIR model given (5.7)-(5.9) using the solver tools developed in the previous chapters. For typical parameter values the models are not stiff, and the explicit RK solvers work well. For instance, a simple code which implements the SIR model as a function, and solves it using the fourth-order RK method, may look as follows:

```
from ODESolver import RungeKutta4
import numpy as np
import matplotlib.pyplot as plt

def SIR_model(t,u):
    beta = 1.0
    nu = 1/7.0
    S, I, R = u[0], u[1], u[2]
    N = S+I+R
    dS = -beta*S*I/N
    dI = beta*S*I/N - nu*I
    dR = nu*I
    return [dS,dI,dR]

S0 = 1000
I0 = 1
R0 = 0

solver= RungeKutta4(SIR_model)
solver.set_initial_condition([S0,I0,R0])
time_points = np.linspace(0, 100, 1001)
t, u = solver.solve(time_points)
S = u[:,0];  I = u[:,1]; R = u[:,2]

plt.plot(t,S,t,I,t,R)
plt.show()
```

The resulting plot is shown in Figure 5.2.

A class implementation of the SIR model. As noted above, estimating the parameters in the model is often challenging. In fact, one of the most important application of models of this kind is to predict the dynamics of new and unknown diseases, for instance during the global Covid19 pandemic. Accurate predictions of the number of disease cases can be extremely important in planning the response to the epidemic, but the challenge is that for a new disease all the parameters are largely unknown. There are ways to estimate the parameters from the early disease dynamics, but the estimates will contain a large degree of uncertainty, and a common strategy is then to run the model for multiple parameter sets to get an idea of the different disease outbreak scenarios that can be expected. We can easily run the code above for multiple values of `beta` and `nu`, but it is inconvenient that both parameters are hardcoded as local variables in the `SIR_model` function, so we need to edit the code for each new parameter value we want. As we have seen earlier, it is much better to represent such a parameterized function as a class, where the parameters can be set in the constructor and the function itself is implemented in a `__call__` method. A class for the SIR model could look like:

```
class SIR:
    def __init__(self, beta, nu):
        self.beta = beta
        self.nu = nu

    def __call__(self,u,t):
        S, I, R = u[0], u[1], u[2]
        N = S+I+R
        dS = -self.beta*S*I/N
        dI = self.beta*S*I/N - self.nu*I
        dR = self.nu*I
        return [dS,dI,dR]
```

As for the models considered in earlier chapters, the use of the class is very similar to the use of the `SIR_model` function above. We create an instance of the class with given values of `beta` and `nu`, and then this instance can be passed to the ODE solver just as any regular Python function.

5.2 Extending the SIR model

The SIR model itself in its simplest form is rarely used for predictive simulations of real-world diseases, but various extensions of the model are used to a large extent. Many such extensions have been derived, in order to best fit the dynamics of different infectious diseases. We will here consider a few such extensions, which are all based on the building blocks of the simple SIR model.

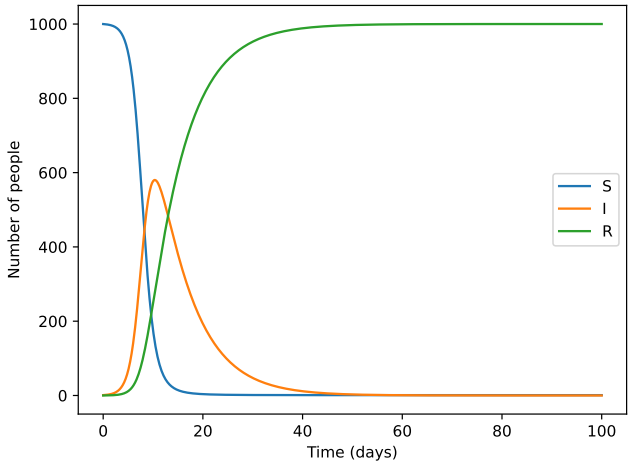

Fig. 5.2 Solution of the simplest version of the SIR model, showing how the number of people in each category (S, I, and R) changes with time.

A SIR model without life-long immunity. One very simple modification of the model above is to remove the assumption of life-long immunity. The model (5.7)-(5.9) describes a one-directional flux towards the R category, and if we solve the model for a sufficiently long time interval the entire population will end up in R. This situation is not realistic for many diseases, since immunity is often lost or reduced with time. In the model this loss can be described by a leakage of people from the R category back to S. If we introduce the parameter γ to describe this flux ($1/\gamma$ being the mean time for immunity), the modified equation system looks like

$$\begin{aligned} S'(t) &= -\beta SI/N + \gamma R, \\ I'(t) &= \beta SI/N - \nu I, \\ R'(t) &= \nu I - \gamma R. \end{aligned}$$

As above, we see that the reduction in R is matched by an increase in S of exactly the same magnitude. The total population $S+I+R$ remains constant. The model can be implemented by a trivial extension of the `SIR` class shown above, by simply adding one additional parameter to the constructor and the extra terms in the `dS` and `dR` equations. Depending on the choice of the parameters, the model may show far more interesting dynamics than the simplest SIR model. An example solution is shown in Figure 5.3. Here, we set $\beta = 0.001, \nu = 1/7.0$, and $\gamma = 1.0/50$, thereby assuming that the mean duration of the disease is seven days and the mean duration of immunity is 50 days.

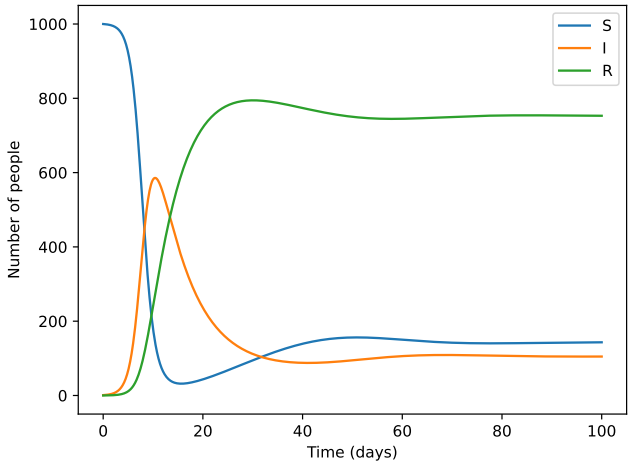

Fig. 5.3 Illustration of a SIR model without lifelong immunity, where people move from the R category back to S after a given time.

A SEIR model to capture the incubation period. For many important infections, there is a significant incubation period during which individuals have been infected, but they are not yet infectious themselves. To capture these dynamics in the model, we may add a category E (for exposed). When people are infected they will then move into the E category rather than directly to I, and then gradually move over to the infected state where they can also infect others. The model for how susceptible people get infected is kept exactly as in the ordinary SIR model. Such a SEIR model is illustrated in Figure 5.4, and the ODEs may look like

$$\begin{aligned} S'(t) &= -\beta SI/N + \gamma R, \\ E'(t) &= \beta SI/N - \mu E, \\ I'(t) &= \mu E - \nu I, \\ R'(t) &= \nu I - \gamma R. \end{aligned}$$

Notice that the overall structure of the model remains the same. Since the total population is conserved, all terms are balanced in the sense that they occur twice in the model, with opposite signs. A decrease in one category is always matched with an identical increase in another category. It is always useful to be aware of such fundamental properties in a model, since they can easily be checked in the computed solutions and may reveal errors in the implementation.

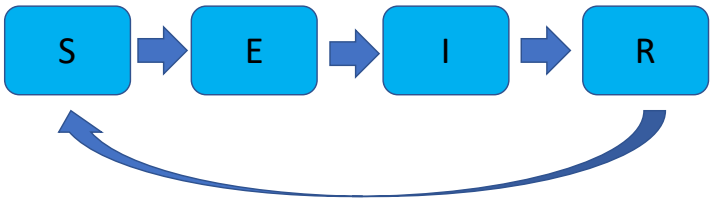

Fig. 5.4 Illustration of the SEIR model, without life-long immunity.

Again, this small extension of the model does not make it much more difficult to solve. The following code shows an example of how the SEIR model can be implemented as a class and solved with the `ODESolver` hierarchy:

```
from ODESolver import RungeKutta4
import numpy as np
import matplotlib.pyplot as plt

class SEIR:
    def __init__(self, beta, mu, nu, gamma):
        self.beta = beta
        self.mu = mu
        self.nu = nu
        self.gamma = gamma


    def __call__(self,u,t):
        S, E, I, R = u
        N = S+I+R+E
        dS = -self.beta*S*I/N + self.gamma*R
        dE  = self.beta*S*I/N - self.mu*E
        dI =  self.mu*E - self.nu*I
        dR = self.nu*I - self.gamma*R
        return [dS,dE,dI,dR]

S0 = 1000
E0 = 0
I0 = 1
R0 = 0
model = SEIR(beta=1.0, mu=1.0/5,nu=1.0/7,gamma=1.0/50)

solver= RungeKutta4(model)
solver.set_initial_condition([S0,E0,I0,R0])
time_points = np.linspace(0, 100, 101)
u, t = solver.solve(time_points)
S = u[:,0]; E = u[:,1]; I = u[:,2]; R = u[:,3]

plt.plot(t,S,t,E,t,I,t,R)
plt.show()
```

5.3 A model of the Covid19 pandemic

The models considered above can be adapted to describe more complex disease behavior by adding more categories of people and possibly more interactions between the different categories. We will now consider an extension of the SEIR model above into a model that was used by Norwegian health authorities to predict the spread of the Covid19 pandemic through 2020 and 2021. We will here derive the model as a system of ODEs, just like the models considered above, while the real model that is used to provide Covid19 predictions for health authorities is a stochastic model.[2] A stochastic model is somewhat more flexible than the deterministic ODE version, and considers the inherent randomness and variability in disease transmission. The model assumes that disease transmission is a stochastic process, meaning that the probability of an individual getting infected is not fixed, but depends on random events and chance encounters with infected individuals. In a stochastic SIR model, the number of individuals in each compartment is modeled using probability distributions rather than deterministic equations, and transitions between compartments are modeled as stochastic processes rather than a continuous flux described by ODEs. One advantage of stochastic models is that one can more easily incorporate dynamics such as model parameters that vary with time after infection. For instance, the infectiousness (β) should typically follow a bell-shaped curve that increases gradually after infection, reaches a peak after a few days, and is then reduced. Such behavior is easier to incorporate in a stochastic model than in the deterministic ODE model considered here, which essentially assumes a constant β for everyone in the I category. However, the overall structure and dynamics of the two model types are exactly the same, and for certain choices of the model parameters the stochastic and deterministic models become equivalent. See, for instance, [6] for a discussion of stochastic and deterministic epidemiology models.

An important characteristic of Covid19 is that people may be infected, and able to infect others, even if they experience no symptoms. This property obviously has a massive impact on the disease spread, since people are unaware that they are infected and therefore take no precautions against infecting others. Two separate groups of asymptomatic yet infectious people have been identified:

- A certain number of people are infected, but never develop any symptoms, or the symptoms are so mild that they are mistaken for other mild airway infections. These asymptomatic people can still infect other, but with a lower infectiousness than the symptomatic group, and they need to be treated as a separate category.
- The other group, which is probably even more important for the disease dynamics, consist of people who are infected and will develop symptoms, but the symptoms have not developed yet. They are, however, still able to

[2] See `https://github.com/folkehelseinstituttet/spread`

infect others, unlike the people in the *exposed* (E) category of the simple SEIR model above.

These two groups can be modeled by adding to new compartments to the SEIR model introduced earlier. We split the exposed category in two, E_1 and E_2, with the first being non-infectious and the second being able to infect others. The I category is also divided in two; a symptomatic I and an asymptomatic I_a. The flux from S to E_1 will be similar to the SEIR model, but from E_1 people will follow one of two possible trajectories. Some will move on to E_2 and then into I and finally R, while others move directly into I_a and then to R. The model is illustrated in Figure 5.5. Since there are two different E-categories and two different I-categories, we refer to the model as a SEEIIR model.

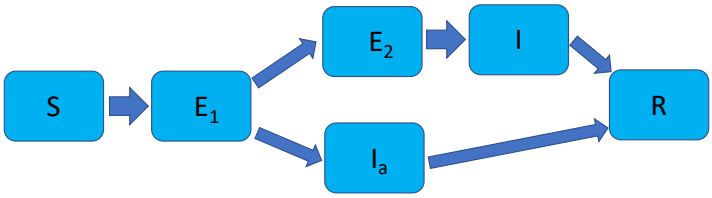

Fig. 5.5 Illustration of the Covid19 epidemic model, with two alternative disease trajectories.

The derivation of the model equations is similar to the simpler models considered above, but there will be more equations as well as more terms in each equation. The most important extension from the models above is that the SEEIIR model has three categories of infectious people; E_2, I, and I_a. All of these interact with the S category to create new infections, and we model each of these interactions exactly as we did above. In a time interval Δt, we have the following three contributions to the flux from S to E_1:

- Infected by people in I: $\beta \Delta t S I/N$.
- Infected by people in I_a: $r_{ia}\beta \Delta t S I_a/N$
- Infected by people in E_2: $r_{e2}\beta \Delta t S E_2/N$

We allow the infectiousness to be different between the three categories, incorporated through a main infectiousness parameter β and two parameters r_{ia}, r_{e2} that scale the infectiousness for the two respective groups. Considering all three contributions, and following the same steps as above to construct a difference equation and then an ODE, we get the following equation for the S category:

$$\frac{dS}{dt} = -\beta \frac{SI}{N} - r_{ia}\beta \frac{SI_a}{N} - r_{e2}\beta \frac{SE_2}{N}. \tag{5.11}$$

When people get infected they move from S to E_1, so the same three terms must appear in the equation for E_1, with opposite signs. Furthermore, people

in E_1 will move either to E_2 or I_a. We have

$$\begin{aligned}\frac{dE_1}{dt} &= \beta\frac{SI}{N} + r_{ia}\beta\frac{SI_a}{N} + r_{e2}\beta\frac{SE_2}{N} - \lambda_1(1-p_a)E_1 - \lambda_1 p_a E_1 \\ &= \beta\frac{SI}{N} + r_{ia}\beta\frac{SI_a}{N} + r_{e2}\beta\frac{SE_2}{N} - \lambda_1 E_1.\end{aligned}$$

Here, p_a is a parameter describing the proportion of infected people that never develop symptoms, while $1/\lambda_1$ is the mean duration of the non-infectious incubation period. The term $\lambda_1(1-p_a)E_1$ represents people moving to E_2, and $\lambda_1 p_a E_1$ are people moving to I_a. In the equation for E_1 we can combine these two fluxes into a single term, but they must be considered separately in the equations for E_2 and I_a.

Moving to the next step in Figure 5.5, we need to consider the two trajectories separately. Starting with the people that develop symptoms, the E_2 compartment will get an influx of people from E_1, and an outflux of people moving on to the infected I category, while I gets an influx from E_2 and an outflux to R. The ODEs for these two categories become

$$\begin{aligned}\frac{dE_2}{dt} &= \lambda_1(1-p_a)E_1 - \lambda_2 E_2, \\ \frac{dI}{t} &= \lambda_2 E_2 - \mu I,\end{aligned}$$

where $1/\lambda_2$ and $1/\mu$ are the mean durations of the E_2 and I phases, respectively. The model for the asymptomatic disease trajectory is somewhat simpler, with I_a receiving an influx from E_1 and losing people directly to R. We have

$$\frac{dI_a}{dt} = \lambda_1 p_a E_1 - \mu I_a,$$

where we have assumed that the duration of the I_a period is the same as for I, i.e. $1/\mu$. Finally, the dynamics of the recovered category are governed by

$$\frac{dR}{dt} = \mu I + \mu I_a.$$

Notice that we do not consider flow from the R category back to S, so we have effectively assumed life-long immunity. This assumption is not correct for Covid19, but in the early phase of the pandemic the duration of immunity was largely unknown, and the loss of immunity was therefore not considered in the models.

To summarize, the complete ODE system of the SEEIIR model can be written as

$$
\begin{aligned}
\frac{dS}{dt} &= -\beta\frac{SI}{N} - r_{ia}\beta\frac{SI_a}{N} - r_{e2}\beta\frac{SE_2}{N},\\
\frac{dE_1}{dt} &= \beta\frac{SI}{N} + r_{ia}\beta\frac{SI_a}{N} + r_{e2}\beta\frac{SE_2}{N} - \lambda_1 E_1,\\
\frac{dE_2}{dt} &= \lambda_1(1-p_a)E_1 - \lambda_2 E_2,\\
\frac{dI}{dt} &= \lambda_2 E_2 - \mu I,\\
\frac{dI_a}{dt} &= \lambda_1 p_a E_1 - \mu I_a,\\
\frac{dR}{dt} &= \mu(I + I_a).
\end{aligned}
$$

A suitable choice of default parameters for the model can be as follows:

Parameter	Value
β	0.33
r_{ia}	0.1
r_{e2}	1.25
λ_1	0.33
λ_2	0.5
p_a	0.4
μ	0.2

These parameters are similar to the ones used by the health authorities to model the early phase of the Covid19 outbreak in Norway. At this time the behavior of the disease was largely unknown, and it was also difficult to estimate the number of disease cases in the population. It was therefore challenging to fit the parameter values, and they were all associated with considerable uncertainty. As mentioned earlier, the hardest parameters to estimate are the ones related to infectiousness and disease spread, which in the present model are β, r_{ia}, and r_{e2}. These have been updated many times through the course of the pandemic, both to reflect new knowledge about the disease and actual changes in disease spread caused by new mutations or changes in the behavior of the population. Notice that we have set $r_{e2} > 1$, which means that people in the E_2 category are more infectious than the infected group in I. This assumption reflects the fact that the E_2 group is asymptomatic, so people in this group are expected to move around more and therefore potentially infect more people than the I group. The I_a group, on the other hand, is also asymptomatic and therefore likely to have normal social interactions, but it is assumed that these people have a very low virus count. They are therefore less infectious than the people that develop symptoms, which is reflected in the low value of r_{ia}.

The parameters μ, λ_1, and λ_2 are given in units of days^{-1}, so the mean duration of the symptomatic disease period is five days $(1/\mu)$, the non-infectious

incubation period lasts three days on average ($1/\lambda_1$), while the mean duration of the infectious incubation period (E_2) is two days ($1/\lambda_2$). In the present model, which has multiple infectious categories, the basic reproduction number is given by

$$R0 = r_{e2}\beta/\lambda_2 + r_{ia}\beta/\mu + \beta/\mu,$$

since the mean durations of the E_2 period is $1/\lambda_2$ and the mean duration of both I and Ia is $1/\mu$. The parameter choices listed above gives $R0 \approx 2.62$, which is the value used by the Institute of Public Health (FHI) to model the early stage of the outbreak in Norway, from mid-February to mid-March 2020.

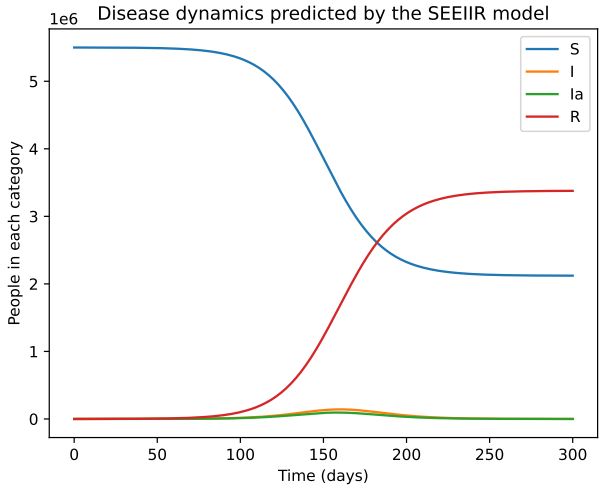

Fig. 5.6 Solution of the SEEIIR model with the default parameter values, which are similar to the values used by Norwegian health authorities during the early phase of the Covid19 pandemic.

Although the present model is somewhat more complex than the previous ones, the implementation is not very different. A class implementation may look as follows:

```
class SEEIIR:
    def __init__(self, beta=0.33, r_ia=0.1,
                 r_e2=1.25, lmbda_1=0.33,
                 lmbda_2=0.5, p_a=0.4, mu=0.2):

        self.beta = beta
        self.r_ia = r_ia
        self.r_e2 = r_e2
        self.lmbda_1 = lmbda_1
        self.lmbda_2 = lmbda_2
```

```
        self.p_a = p_a
        self.mu = mu

    def __call__(self, t, u):
        beta = self.beta
        r_ia = self.r_ia
        r_e2 = self.r_e2
        lmbda_1 = self.lmbda_1
        lmbda_2 = self.lmbda_2
        p_a = self.p_a
        mu = self.mu

        S, E1, E2, I, Ia, R = u
        N = sum(u)
        dS = -beta * S * I / N - r_ia * beta * S * Ia / N \
            - r_e2 * beta * S * E2 / N
        dE1 = beta * S * I / N + r_ia * beta * S * Ia / N \
            + r_e2 * beta * S * E2 / N - lmbda_1 * E1
        dE2 = lmbda_1 * (1 - p_a) * E1 - lmbda_2 * E2
        dI = lmbda_2 * E2 - mu * I
        dIa = lmbda_1 * p_a * E1 - mu * Ia
        dR = mu * (I + Ia)
        return [dS, dE1, dE2, dI, dIa, dR]
```

The model can be solved with any of the methods in the `ODESolver` hierarchy, just as the simpler models considered earlier. An example solution with the default parameter values is shown in Figure 5.6. Since the parameters listed above are based on the very first stage of the pandemic, when no restrictions were in place, this solution may be interpreted as a potential worst case scenario for the pandemic in Norway if no restrictions were imposed by the government. While the plot for the I category does not look too dramatic, a closer inspection reveals that the peak is at just above 140,000 people. Considering what was known, and, more importantly, what was not known, about the severity of Covid19 at that stage, it is not surprising that a scenario of 140,000 people being infected simultaneously caused some alarm with the health authorities. Another interesting observation from the curve is that the S category flattens out well below the total population number. This behavior is an example of so-called herd immunity, that when a sufficient number of people are immune to the disease, it will effectively stop spreading even if a large number of people are still susceptible. As we all know, severe restrictions were put in place in most countries during the early spring of 2020, which makes it impossible to know whether this worst case scenario would ever materialize. If we want to match the actual dynamics of the pandemic in Norway, we would need to incorporate the effect of societal changes and altered infectiousness by making the β parameter a function of time. For instance, we can define it as a piecewise constant function to match the reproduction numbers estimated and published by the health authorities.

Appendix A
Programming of difference equations

Although the main focus of these notes is on solvers for *differential equations*, we find it useful to include a chapter on the closely related class of problems known as *difference equations*. The main motivation for including this topic in a book on ODEs is to highlight the similarity between the two classes of problems, and in particular the similarity of the solution methods and their implementation. Indeed, solving ODEs numerically can be seen as a two-step procedure. First, a numerical method is applied to turn *differential equations* into *difference equations*, and then these equations are solved using simple for-loop. The standard formulation of difference equations is very easy to translate into a computer program, and some readers may find it easier to study these equations first, before moving on to ODEs. In the present chapter we will also touch upon famous sequences and series, which have important applications both in the numerical solution of ODEs and elsewhere.

A.1 Sequences and Difference Equations

Sequences is a central topic in mathematics, which has important applications in numerical analysis and scientific computing. In the most general sense, a sequence is simply a collection of numbers:

$$x_0,\ x_1,\ x_2,\ \ldots,\ x_n,\ldots.$$

For some sequences we can derive a formula that gives the the n-th number x_n as a function of n. For instance, all the odd numbers form a sequence

$$1,3,5,7,\ldots,$$

and for this sequence we can write a simple formula for the n-th term;

$$x_n = 2n+1.$$

With this formula at hand, the complete sequence can be written on a compact form;

$$(x_n)_{n=0}^{\infty}, \quad x_n = 2n+1.$$

Other examples of sequences include

$$1,\ 4,\ 9,\ 16,\ 25,\ \ldots \quad (x_n)_{n=0}^{\infty},\ x_n = n^2,$$
$$1,\ \frac{1}{2},\ \frac{1}{3},\ \frac{1}{4},\ \ldots \quad (x_n)_{n=0}^{\infty},\ x_n = \frac{1}{n+1},$$
$$1,\ 1,\ 2,\ 6,\ 24,\ \ldots \quad (x_n)_{n=0}^{\infty},\ x_n = n!,$$
$$1,\ 1+x,\ 1+x+\frac{1}{2}x^2,\ 1+x+\frac{1}{2}x^2+\frac{1}{6}x^3,\ \ldots \quad (x_n)_{n=0}^{\infty},\ x_n = \sum_{j=0}^{n}\frac{x^j}{j!}.$$

These are all formulated as infinite sequences, which is common in mathematics, but in real-life applications sequences are usually finite: $(x_n)_{n=0}^{N}$. Some familiar examples include the annual value of a loan or an investment.

In many cases it is impossible to derive an explicit formula for the entire sequence, and x_n is instead given by a relation involving x_{n-1} and possibly earlier terms. Such equations are called *difference equations*, and they can be challenging to solve with analytical methods, since in order to compute the n-th term of a sequence we need to compute the entire sequence $x_0, x_1, \ldots, x_{n-1}$. This can be very tedious to do by hand or using a calculator, but on a computer the equation is easy to solve with a for-loop. Combining sequences and difference equations with programming therefore enables us to consider far more interesting and useful cases.

A difference equation for computing interest. To start with a simple example, consider the problem of computing how an invested sum of money grows over time. In its simplest form, this problem can be written as putting x_0 money in a bank at year 0, with interest rate p percent per year. What is then the value after n years? If p is constant, the solution to this problem is given by the simple formula

$$x_n = x_0(1+p/100)^n,$$

so there is really no need to formulate and solve the problem as a difference equation. However, very simple generalizations, such as a non-constant interest rate, makes this formula difficult to apply, while a formulation based on a difference equation will still be applicable. To formulate the problem as a difference equation, we use the fact that the amount x_{n+1} at year $n+1$ is simply the amount at year n plus the interest for year n. This gives the following relation between x_{n+1} and x_n:

$$x_{n+1} = x_n + \frac{p}{100}x_n.$$

In order to compute x_n, we can now simply start with the known x_0, and compute $x_1, x_2, \ldots, x_n$. The procedure involves repeating a simple calculation many times, which is tedious to do by hand, but very well suited for a computer. The complete program for solving this difference equation may look like:

```
import numpy as np
import matplotlib.pyplot as plt
x0 = 100                        # initial amount
p = 5                           # interest rate
N = 4                           # number of years
x = np.zeros(N+1)

x[0] = x0
for n in range(1,N+1):
    x[n] = x[n-1] + (p/100.0)*x[n-1]

plt.plot(x, 'ro')
plt.xlabel('years')
plt.ylabel('amount')
plt.show()
```

The three lines starting with `x[0] = x0` make up the core of the program. We here initialize the first element in our solution array with the known `x0`, and then step into the for-loop to compute the rest. The loop variable `n` runs from 1 to $N(=4)$, and the formula inside the loop computes `x[n]` from the known `x[n-1]`. Notice also that we pass a single array as argument to `plt.plot`, while in most of the examples in this book we sent two; typically representing time on the x-axis and the solution on the y-axis. When only one array of numbers is sent to `plot`, these are automatically interpreted as the y-coordinates of the points, and the x-coordinates will simply be the indices of the array, in this case the numbers from 0 to N.

Solving a difference equation without using arrays. The program above stored the sequence as an array, which is a convenient way to program the solver and enables us to plot the entire sequence. However, if we are only interested in the solution at a single point, i.e., x_n, there is no need to store the entire sequence. Since each x_n only depends on the previous value x_{n-1}, we only need to store the last two values in memory. A complete loop can look like this:

```
x_old = x0
for n in index_set[1:]:
    x_new = x_old + (p/100.)*x_old
    x_old = x_new  # x_new becomes x_old at next step
print('Final amount: ', x_new)
```

For this simple case we can actually make the code even shorter, since `x_old` is only used in a single line and we can just as well overwrite it once it has been used:

```
x = x0          #x is here a single number, not array
for n in index_set[1:]:
    x = x + (p/100.)*x
print('Final amount: ', x)
```

We see that these codes store just one or two numbers, and for each pass through the loop we simply update these and overwrite the values we no longer need. Although this approach is quite simple, and we obviously save some memory since we do not store the complete array, programming with an array `x[n]` is usually safer, and we are often interested in plotting the entire sequence. We will therefore mostly use arrays in the subsequent examples.

Extending the solver for the growth of money. Say we are interested in changing our model for interest rate, to a model where the interest is added every day instead of every year. The interest rate per day is $r = p/D$ if p is the annual interest rate and D is the number of days in a year. A common model in business applies $D = 360$, but n counts exact (all) days. The difference equation relating one day's amount to the previous is the same as above

$$x_n = x_{n-1} + \frac{r}{100}x_{n-1},$$

except that the yearly interest rate has been replaced by the daily (r). If we are interested in how much the money grows from a given date to another we also need to find the number of days between those dates. This calculation could of course be done by hand, but Python has a convenient module named `datetime` for this purpose. The following session illustrates how it can be used:

```
>>> import datetime
>>> date1 = datetime.date(2017, 9, 29)  # Sep 29, 2017
>>> date2 = datetime.date(2018, 8, 4)  # Aug 4, 2018
>>> diff = date2 - date1
>>> print(diff.days)
309
```

Putting these tools together, a complete program for daily interest rates may look like

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

x0 = 100                            # initial amount
p = 5                               # annual interest rate
r = p/360.0                         # daily interest rate

date1 = datetime.date(2017, 9, 29)
date2 = datetime.date(2018, 8, 4)
diff = date2 - date1
N = diff.days
index_set = range(N+1)
```

```
x = np.zeros(len(index_set))

x[0] = x0
for n in index_set[1:]:
    x[n] = x[n-1] + (r/100.0)*x[n-1]

plt.plot(index_set, x)
plt.xlabel('days')
plt.ylabel('amount')
plt.show()
```

This program is slightly more sophisticated than the first one, but one may still argue that solving this problem with a difference equation is unnecessarily complex, since we could just apply the well-known formula $x_n = x_0(1+\frac{r}{100})^n$ to compute any x_n we want. However, we know that interest rates change quite often, and the formula is only valid for a constant r. For the program based on solving the difference equation, on the other hand, only minor changes are needed in the program to handle a varying interest rate. The simplest approach is to let `p` be an array of the same length as the number of days, and fill it with the correct interest rates for each day. The modifications to the program above may look like this:

```
p = np.zeros(len(index_set))
# fill p[n] with correct values

r = p/360.0                      # daily interest rate
x = np.zeros(len(index_set))

x[0] = x0
for n in index_set[1:]:
    x[n] = x[n-1] + (r[n-1]/100.0)*x[n-1]
```

The only real difference from the previous example is that we initialize `p` as an array, and then `r = p/360.0` becomes an array of the same length. In the formula inside the for-loop we then look up the correct value `r[n-1]` for each iteration of the loop. Filling `p` with the correct values can be non-trivial, but many cases can be handled quite easily. For instance, say the interest rate is piecewise constant and increases from 4.0% to 5.0% on a given date. Code for filling the array with values may then look like this

```
date0 = datetime.date(2017, 9, 29)
date1 = datetime.date(2018, 2, 6)
date2 = datetime.date(2018, 8, 4)
Np = (date1-date0).days
N = (date2-date0).days

p = np.zeros(len(index_set))
p[:Np] = 4.0
p[Np:] = 5.0
```

A.2 More Examples of Difference Equations

As noted above, sequences, series, and difference equations have countless applications in mathematics, science, and engineering. Here we present a selection of well known examples.

Fibonacci numbers as a difference equation. The sequence defined by the difference equation

$$x_n = x_{n-1} + x_{n-2}, \quad x_0 = 1,\ x_1 = 1.$$

is called the Fibonacci numbers. It was originally derived for modeling rat populations, but it has a range of interesting mathematical properties and has therefore attracted much attention from mathematicians. The equation for the Fibonacci numbers differs from the previous ones, since x_n depends on the two previous values $(n-1,\ n-2)$, which makes this a *second order difference equation*. This classification is important for mathematical solution techniques, but in a program the difference between first and second order equations is small. A complete code to solve the difference equation and generate the Fibonacci numbers can be written as

```
import sys
from numpy import zeros

N = int(sys.argv[1])
x = zeros(N+1, int)
x[0] = 1
x[1] = 1
for n in range(2, N+1):
    x[n] = x[n-1] + x[n-2]
    print(n, x[n])
```

We use the builtin list `sys.argv` from the `sys` model in order to provide the input `N` as a command line argument. See, for instance, [16] for an explanation. Notice that in this case we need to initialize both `x[0]` and `x[1]` before starting the loop, since the update formula involves both `x[n-1]` and `x[n-2]`. This is the main difference between this second order equation and the programs for first order equations considered above. The Fibonacci numbers grow quickly and running this program for large N will lead to problems with overflow (try for instance $N = 100$). The NumPy `int` type supports up to 9223372036854775807, which is almost 10^{19}, so this is very rarely a problem in practical applications. We can fix the problem by avoiding NumPy arrays and instead use the standard Python `int` type, but we will not go into these details here.

Logistic growth. If we return to the initial problem of calculating growth of money in a bank, we can write the classical solution formula more compactly as

$$x_n = x_0(1 + p/100)^n = x_0 C^n \quad (= x_0 e^{n \ln C}),$$

with $C = (1 + p/100)$. Since n counts years, this is an example of exponential growth in time, with the general formula $x = x_0 e^{\lambda t}$. Populations of humans, animals, and other organisms also exhibit the same type of growth when there are unlimited resources (space and food), and the model for exponential growth therefore has many applications in biology.[1] However, most environments can only support a finite number R of individuals, while in the exponential growth model the population will continue to grow indefinitely. How can we alter the equation to be a more realistic model for growing populations?

Initially, when resources are abundant, we want the growth to be exponential, i.e., to grow with a given rate $r\%$ per year according to the difference equation introduced above:

$$x_n = x_{n-1} + (r/100)x_{n-1}.$$

To enforce the growth limit as $x_n \to R$, r must decay to zero as x_n approaches R. The simplest variation of $r(n)$ is linear:

$$r(n) = \varrho\left(1 - \frac{x_n}{R}\right)$$

We observe that $r(n) \approx \varrho$ for small n, when $x_n \ll R$, and $r(n) \to 0$ as n grows and $x_n \to R$. This formulation of the growth rate leads to the logistic growth model:

$$x_n = x_{n-1} + \frac{\varrho}{100}x_{n-1}\left(1 - \frac{x_{n-1}}{R}\right).$$

This is a *nonlinear* difference equation, while all the examples considered above were linear. The distinction between linear and non-linear equations is very important for mathematical analysis of the equations, but it does not make much difference when solving the equation in a program. To modify the interest rate program above to describe logistic growth, we can simply replace the line

```
x[n] = x[n-1] + (p/100.0)*x[n-1]
```

by

```
x[n] = x[n-1] + (rho/100)*x[n-1]*(1 - x[n-1]/R)
```

A complete program may look like

```
import numpy as np
import matplotlib.pyplot as plt
x0 = 100                        # initial population
rho = 5                         # growth rate in %
R = 500                         # max population (carrying capacity)
```

[1] As discussed in Chapter 1, the formula $x = x_0 e^{\lambda t}$ is the solution of the differential equation $dx/dt = \lambda x$, which illustrates the close relation between difference equations and differential equations.

```
N = 200                          # number of years

index_set = range(N+1)
x = np.zeros(len(index_set))

x[0] = x0
for n in index_set[1:]:
    x[n] = x[n-1] + (rho/100) *x[n-1]*(1 - x[n-1]/R)

plt.plot(index_set, x)
plt.xlabel('years')
plt.ylabel('amount')
plt.show()
```

Note that the logistic growth model is more commonly formulated as an ODE, which we considered in Chapter 1 For certain choices of numerical method and discretization parameters, the program for solving the ODE is identical to the program for the difference equation considered here.

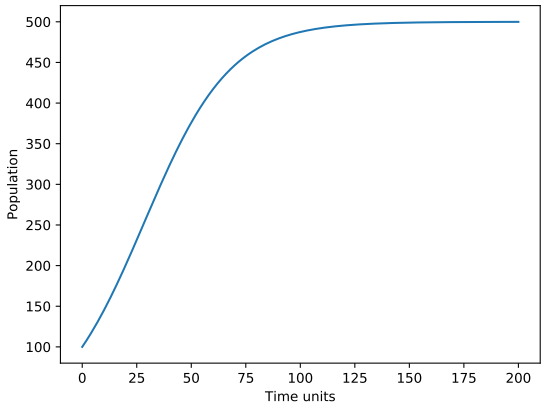

Fig. A.1 Solution of the logistic growth model for $x_0 = 100, \rho = 5.0, R = 500$.

The factorial as a difference equation. The factorial $n!$ is defined as

$$n! = n(n-1)(n-2)\cdots 1, \quad 0! = 1 \tag{A.1}$$

The following difference equation has $x_n = n!$ as solution and can be used to compute the factorial:

$$x_n = nx_{n-1}, \quad x_0 = 1$$

As above, a natural question to ask is whether such a difference equation is very useful, when we can simply use the formula (A.1) to compute the factorial for any value of n. One answer to this question is that in many

applications, some of which will be considered below, we need to compute the entire sequence of factorials $x_n = n!$ for $n = 0, \ldots N$. We could still apply (A.1) to compute each one, but it involves a lot of redundant computations, since we perform n multiplications for each new x_n. When solving the difference equation, each new x_n requires only a single multiplication, and for large values of n this may speed up the program considerably.

Newton's method as a difference equation. Newton's method is a popular method for solving non-linear equations on the form

$$f(x) = 0.$$

Starting from some initial guess x_0, Newton's method gradually improves the approximation by iterations

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

We may now recognize this as nonlinear first-order difference equation. As $n \to \infty$, we hope that $x_n \to x_s$, where x_s is the solution to $f(x_s) = 0$. In practice we solve the equation for $n \leq N$, for some finite N, just as for the difference equations considered earlier. But how do we choose N so that x_N is sufficiently close to the true solution x_s? Since we want to solve $f(x) = 0$, the best approach is to solve the equation until $f(x) \leq \epsilon$, where ϵ is a small tolerance. In practice, Newton's method will usually converge rather quickly, or not converge at all, so setting some upper bound on the number of iterations is a good idea. A simple implementation of Newton's method as a Python function may look like

```
def Newton(f, dfdx, x, epsilon=1.0E-7, max_n=100):
    n = 0
    while abs(f(x)) > epsilon and n <= max_n:
        x = x - f(x)/dfdx(x)
        n += 1
    return x, n, f(x)
```

The arguments to the function are Python functions `f` and `dfdx` implementing $f(x)$ and its derivative. Both of these arguments are called inside the function and must therefore be callable. The `x` argument is the initial guess for the solution x, and the two optional arguments at the end are the tolerance and the maximum number of iteration. Although the method is implemented as a while-loop rather than a for-loop, the main structure of the algorithm is exactly the same as for the other difference equations considered earlier.

A.3 Systems of Difference Equations

All the examples considered so far have been scalar difference equations, which describe how a single quantity changes from one step to the next. However, many applications require tracking multiple variables simultaneously, and dynamics of these variables may be coupled in the sense that the value of one variable at step n depends on the value of multiple variables at step $n-1$. As an example we may consider a simple extension of the interest rate model considered earlier. Assume that we have a fortune F invested with an annual interest rate of p percent, just as above, but now we also want to consume an amount c_n every year. We model for computing our fortune x_n at year n can be formulated as a small extension of the difference equation considered earlier. First, simple reasoning tells us that the fortune at year n is equal to the fortune at year $n-1$ plus the interest minus the amount we spent in year $n-1$. We have

$$x_n = x_{n-1} + \frac{p}{100}x_{n-1} - c_{n-1}.$$

In the simplest case we can assume that c_n is constant, which would make this model a trivial extension of the interest rate model considered earlier. However, it is more natural to let c_n increase because of inflation, and in this case we obtain a system of difference equations describing the evolution of x_n and c_n. For instance, we may assume that c_n should grow with a rate of I percent per year, and in the first year we want to consume q percent of first year's interest. The governing system of difference equations then becomes

$$\begin{aligned} x_n &= x_{n-1} + \frac{p}{100}x_{n-1} - c_{n-1}, \\ c_n &= c_{n-1} + \frac{I}{100}c_{n-1}. \end{aligned}$$

with initial conditions $x_0 = F$ and $c_0 = (pF/100)(q/100) = \frac{pFq}{10000}$. This is a coupled system of two first order difference equations, but the programming is not much more difficult than for the single equation above. We simply create two arrays `x` and `c`, initialize `x[0]` and `c[0]` to the given initial conditions, and then update `x[n]` and `c[n]` inside the loop. A complete code may look like

```
import numpy as np
import matplotlib.pyplot as plt
F = 1e7                       # initial amount
p = 5                          # interest rate
I = 3
q = 75
N = 40                          # number of years
index_set = range(N+1)
x = np.zeros(len(index_set))
```

```
c = np.zeros_like(x)

x[0] = F
c[0] = q*p*F*1e-4

for n in index_set[1:]:
    x[n] = x[n-1] + (p/100.0)*x[n-1] - c[n-1]
    c[n] = c[n-1] + (I/100.0)*c[n-1]

plt.plot(index_set, x, 'ro',label = 'Fortune')
plt.plot(index_set, c, 'go', label = 'Yearly consume')
plt.xlabel('years')
plt.ylabel('amounts')
plt.legend()
plt.show()
```

As another example of a system of difference equations, we may consider an extension of the logistic growth model considered above. While the logistic model describes the growth of a single population in the absence of predators, the famous Lotke-Volterra model describes the interaction of two species, a predator and a prey, in the same ecosystem. If we let x_n be the number of prey and y_n the number of predators on day n, the model for the population dynamics can be written as

$$x_n = x_{n-1} + ax_{n-1} - bx_{n-1}y_{n-1},$$
$$y_n = y_{n-1} + dbx_{n-1}y_{n-1} - cy_{n-1}.$$

Here, a is the natural growth rate of the prey in the absence of predators, b is the death rate of prey per encounter of prey and predator, c is the natural death rate of predators in the absence of food (prey), and d is the efficiency of turning predated prey into predators. This is a system of two first-order difference equations, just as the previous example, and a complete solution code may look as follows.

```
import numpy as np
import matplotlib.pyplot as plt
x0 = 100                # initial prey population
y0 = 8                  # initial predator pop.
a = 0.0015
b = 0.0003
c = 0.006
d = 0.5
N = 10000               # number of time units (days)
index_set = range(N+1)
x = np.zeros(len(index_set))
y = np.zeros_like(x)

print(x.shape)

x[0] = x0
y[0] = y0
```

```
for n in index_set[1:]:
    x[n] = x[n-1] + a*x[n-1] - b*x[n-1]*y[n-1]
    y[n] = y[n-1] + d*b*x[n-1]*y[n-1] - c*y[n-1]

plt.plot(index_set, x,label = 'Prey')
plt.plot(index_set, y, label = 'Predator')
plt.xlabel('Time')
plt.ylabel('Population')
plt.legend()
plt.show()
```

A.4 Taylor Series and Approximations

One extremely important use of sequences and series is for approximating functions. For instance, commonly used functions such as $\sin x, \ln x$, and e^x have been defined to have some desired mathematical properties, and we have an intuitive understanding of how they look, but we need some kind of algorithm to evaluate the function values. A convenient approach would be to approximate these functions by polynomials, since they are easy to calculate. It turns out that such polynomial approximations exist, and they have been used for centuries to compute exponentials, trigonometric, and other functions. By far, the most famous and widely used series of this kind are the Taylor series, discovered in 1715:

$$f(x) = \sum_{k=0}^{\infty} \frac{1}{k!} \left(\frac{d^k f(0)}{dx^k}\right) x^k. \tag{A.2}$$

Here, the notation $d^k f(0)/dx^k$ means the k-th derivative of f evaluated at $x = 0$. We can calculate a few of the terms in the sum to get

$$f(x) = f(0) + f'(0)x + \frac{1}{2} f''(0)x^2 + \frac{1}{6} f'''(0)x^3 \ldots,$$

which makes it obvious that the right hand side of (A.2) is in fact a polynomial in x. Taylor's result means that for any function $f(x)$, if we can compute the function value and its derivatives for $x = 0$, we can approximate the function value at any x by evaluating a polynomial. For practical applications, we always work with a truncated version of the Taylor series:

$$f(x) \approx \sum_{k=0}^{N} \frac{1}{k!} \left(\frac{d^k f(0)}{dx^k}\right) x^k. \tag{A.3}$$

The approximation improves as N is increased, but the most popular choice is actually $N = 1$, which gives a reasonable approximation close to $x = 0$ and has been essential in developing physics and technology. By a shift of variables we can make these truncated Taylor series accurate around any value $x = a$:

$$f(x) \approx \sum_{k=0}^{N} \frac{1}{k!}(\frac{d^k f(a)}{dx^k})(x-a)^k.$$

One of many applications of truncated Taylor series is to derive numerical methods for ODEs, and to analyze their accuracy, as we briefly introduced in Chapter 2.

As a specific example of a Taylor series, consider the exponential function, where we know that $d^k e^x/dx^k = e^x$ for all k, and $e^0 = 1$. Inserting this into (A.3) yields

$$\begin{aligned} e^x &= \sum_{k=0}^{\infty} \frac{x^k}{k!} \\ &\approx \sum_{k=0}^{N} \frac{x^k}{k!}. \end{aligned}$$

Choosing, for instance, $N = 1$ and $N = 4$, we get

$$\begin{aligned} e^x &\approx 1 + x, \\ e^x &\approx 1 + x + \frac{1}{2}x^2 + \frac{1}{6}x^3, \end{aligned}$$

respectively. These approximations are obviously not very accurate for large x, but close to $x = 0$ they are sufficiently accurate for many applications. Taylor series approximations for other functions can be constructed by similar arguments. Consider, for instance, $sin(x)$, where the derivatives follow the repetitive pattern $\sin'(x) = \cos(x), \sin''(x) = -sin(x), \sin'''(x) = -\cos(x), \ldots\ldots$ We also have $\sin(0) = 0, \cos(0) = 1$, so in general we have $d^k \sin(0)/dx^k = (-1)^k mod(k, 2)$, where $mod(k, 2)$ is zero for k even and

$$\sin x = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k+1}}{(2k+1)!}.$$

Taylor series formulated as difference equations. We consider again the Taylor series for e^x around $x = 0$, given by

$$e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!}.$$

If we now define e_n as the approximation with n terms, i.e. for $k = 0, \ldots, n-1$, we have

$$e_n = \sum_{k=0}^{n-1} \frac{x^k}{k!} = \sum_{k=0}^{n-2} \frac{x^k}{k!} + \frac{x^{n-1}}{(n-1)!},$$

and we can formulate the sum in e_n as the difference equation

$$e_n = e_{n-1} + \frac{x^{n-1}}{(n-1)!}, \quad e_0 = 0. \tag{A.4}$$

We see that this difference equation involves $(n-1)!$, and computing the complete factorial for every iteration involves a large number of redundant multiplications. Above we introduced a difference equation for the factorial, and this idea can be utilized to formulate a more efficient computation of the Taylor polynomial. We have that

$$\frac{x^n}{n!} = \frac{x^{n-1}}{(n-1)!} \cdot \frac{x}{n},$$

and if we let $a_n = x^n/n!$ it can be computed efficiently by solving

$$a_n = a_{n-1} \frac{x}{n}, \quad a_0 = 1.$$

Now we can formulate a system of two difference equations for the Taylor polynomial, where we update each term via the a_n equation and sum the terms via the e_n equation:

$$\begin{aligned} e_n &= e_{n-1} + a_{n-1}, \quad e_0 = 0, \\ a_n &= \frac{x}{n} a_{n-1}, \quad a_0 = 1. \end{aligned}$$

Although we are here solving a system of two difference equations, the computation is far more efficient than solving the single equation in (A.4) directly, since we avoid the repeated multiplications involved in the factorial computation.

A complete Python code for solving the system of difference equation and computing the Taylor approximation to the exponential function may look like

```
import numpy as np

x = 0.5 #approximate exp(x) for x = 0.5

N = 5
index_set = range(N+1)
a = np.zeros(len(index_set))
e = np.zeros(len(index_set))
a[0] = 1
```

```
print(f'Exact: exp({x}) = {np.exp(x)}')
for n in index_set[1:]:
    e[n] = e[n-1] + a[n-1]
    a[n] = x/n*a[n-1]
    print(f'n = {n}, approx. {e[n]}, error = {np.abs(e[n]-np.exp(x)):4.5f}')
```

```
Exact: exp(0.5) = 1.64872
n = 1, approx. 1.00000, error = 0.64872
n = 2, approx. 1.50000, error = 0.14872
n = 3, approx. 1.62500, error = 0.02372
n = 4, approx. 1.64583, error = 0.00289
n = 5, approx. 1.64844, error = 0.00028
```

This small program first prints the exact value e^x for $x = 0.5$, and then the Taylor approximation and associated error for $n = 1$ to $n = 5$. The Taylor series approximation is most accurate close to $x = 0$, so choosing a larger value of x leads to larger errors, and we need to also increase n for the approximation to be accurate.

References

[1] U. M. Ascher and L. R. Petzold. *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equtions*. SIAM, 1998.

[2] C. F. Curtiss and J. O. Hirschfelder. Integration of stiff equations. *Proc. Nat. Acad. Sci.*, 38:235–243, 1952.

[3] Peter Deuflhard and Folkmar Bornemann. *Scientific Computing With Ordinary Differential Equations*, volume 42. Springer, 2012.

[4] J. R. Dormand and P. J. Prince. A family of embedded runge-kutta formulae. *J. Comput. Appl. Math.*, 6:19–26, 1980.

[5] E. Fehlberg. Klassische runge-kutta-formeln vierter und niedrigerer ordnung mit schrittweiten-kontrolle und ihre anwendung auf wärmeleitungsprobleme. *Computing*, 6(1):61–71, 1970.

[6] Priscilla E. Greenwood and Luis F. Gordillo. *Stochastic Epidemic Modeling*, pages 31–52. Springer, 2009.

[7] Kjell Gustafsson. Using control theory to improve stepsize selection in numerical integration of ODE. *IFAC Proceedings Volumes*, 23(8):405–410, 1990.

[8] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I, Nonstiff Problems*. Springer, 1991.

[9] E. Hairer and G. Wanner. *Solving Ordinary Differential Equations II, Stiff and Differential Algebraic Problems*. Springer, 1991.

[10] A.L. Hodgkin and A. F. Huxley. A quantitative description of of membrane current and its aplication to conduction and excitation in nerve. *J Physiol*, 117:500–544, 1952.

[11] J. Keener and J. Sneyd. *Mathematical Physiology*. Springer, 2009.

[12] WO Kermack and AG McKendrick. Contributions to the mathematical theory of epidemics-i. 1927. *Bulletin of mathematical biology*, 53(1-2), 1991.

[13] Anne Kværnø. Singly diagonally implicit runge-kutta methods with an explicit first stage. *BIT Numerical Mathematics*, 44:489–502, 2004.

[14] Hans Petter Langtangen and Hans Petter Langtangen. *A Primer on Scientific Programming With Python*, volume 6. Springer, 2012.

[15] S. Rush and H. Larsen. A practical algorithm for solving dynamic membrane equations. *IEEE Transactions on Biomedical Engineering*, 25(4):389–392, 1978.

[16] Joakim Sundnes. *Introduction to Scientific Programming With Python.* Springer, 2020.

Index