Joakim Sundnes*

Solving Ordinary Differential Equations in Python

Jul 31, 2020

*Simula Research Laboratory and Department of Informatics, University of Oslo.

Preface

These lecture notes are based on the book *A Primer on Scientific Programming with Python* by Hans Petter Langtangen[1], and primarily cover topics from Appendix A, C, and E. The notes are intended as a brief and gentle introduction to solving differential equations in Python, for use in the course *Introduction to programming for scientific applications* (IN1900) at the University of Oslo. To read these notes one should have basic knowledge of Python and NumPy[2], and it is also useful to have a fundamental understanding of ordinary differential equations (ODEs).

The purpose of these notes is to provide a foundation for writing your own ODE solvers in Python. One may ask why this is useful, since 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 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 learn this 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.

Although the main purpose of these notes is to solve differential equations, the topic of the first chapter is *difference equations*. The motivation for this somewhat unusual approach is that, from a programming perspective, difference equations are easier to solve, and a natural step on the way towards solving ODEs. The standard formulation of difference equations in mathematical textbooks is already in a "computer-friendly" form, and is very easy to translate into a Python program using for-loops and arrays. Furthermore, as we shall see in Chapter 2, applying a numerical method such as the For-

[1] Hans Petter Langtangen, *A Primer on Scientific Programming with Python*, 5th edition, Springer-Verlag, 2016.

[2] See for instance Joakim Sundnes, *Introduction to Scientific Programming with Python*, Springer-Verlag, 2020.

ward Euler scheme to an ODE effectively turns the differential equation into a difference equation. If we already know how to program difference equations it is therefore very easy to solve an ODE, by simply adding one extra step at the start of the process. However, although this structure provides a fairly natural step-by-step introduction to ODE programming, it is entirely possible to skip Chapter 1 completely and jump straight into the programming of ODE solvers in Chapter 2.

July 2020 *Joakim Sundnes*

Contents

Chapter 1
Programming of difference equations

Although the main focus of these notes is on solvers for *differential equations*, this first chapter is devoted to the closely related class of problems known as *difference equations*. The main motivation for introducing this topic first is that the mathematical formulation of difference equations is very easy to translate into a computer program. When we move on to ODEs in the next chapter, we shall see that such equations are typically solved by applying some numerical scheme to turn the differential equation into a difference equation, which is then solved. Knowing how to program difference equations is therefore an essential part of solving ODEs.

1.1 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 inifite 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 x_{n-2}. 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? You may recall from earlier in IN1900 (and from high school mathematics) that 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 at year n is simply the amount at year $n-1$ plus the interest for year $n-1$. This gives the following relation between x_n and x_{n-1}:

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

If we want 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 com-

puter. 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
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] + (p/100.0)*x[n-1]

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

This code only involves tools that we have introduced earlier in the course. Notice that we store the result of the call to `range` in a variable `index_set`, which will be a list-like object of numbers from 0 to N (representing the years). We could also have called `range` directly everywhere we need it, but since we use it in three places in the code it is more convenient to use a variable. The core of the program is the three lines starting with `x[0] = x0`. 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]`.

An alternative formulation of the for-loop would be

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

Here `n` runs from 0 to 3, and all the indices inside the loop have been increased by one so that the end result is the same. In this 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 some of the sequence elements are never computed. Such mistakes are probably the most common type of programming error when solving difference equations, and it is a good habit to always examine the loop formulation carefully. If an `IndexError` (or another suspected loop error) occurs, a good debugging strategy is to look at the loop definition to find the lower and upper value of the loop variable (here `n`), and insert both by hand into the formulas inside the loop to check that they make sense. As an example, consider the deliberately wrong code

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

Assuming that the rest of the code is unchanged from the example above, the loop variable `n` will run from 1 to 4. If we first insert the lower bound `n=1` into the formula, we find that the first pass of the loop will try to compute `x[2]` from `x[1]`. However, we have only initialized `x[0]`, so `x[1]` is zero, and therefore `x[2]` and all subsequent values will be set to zero. Furthermore, if we insert the upper bound `n=4` we see that the formula will try to access `x[5]`, but this does not exist and we get an `IndexError`. Performing such simple analysis of a for-loop is often a good way to reveal the source of the error and give an idea of how it can be fixed.

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
```

1.2 Systems of Difference Equations

To consider a related example to the one above, assume that we have a fortune F invested with an annual interest rate of p percent. Every year we plan to consume an amount c_n, where n counts years, and we want to compute our fortune x_n at year n. The problem can be formulated as a difference equation, where the fundamental relation from one year to the next is

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

In the simplest case c_n is constant, but inflation demands c_n to increase. To solve this problem, we 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 extension of the difference equation above 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 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()
```

1.3 More Examples of Difference Equations

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

Fibonacci numbers as a difference equation. No programming or math course is complete without an example on Fibonacci numbers:

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

This sequence was originally derived by Fibonacci for modeling rat populations, but it has a range of peculiar mathematical properties and has therefore attracted much attention from mathematicians. The equation 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. The complete code to solve this 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])
```

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 (try for instance $N = 100$) will lead to problems with overflow. The NumPy `int` type supports up to 9223372036854775807 (almost 10^{19}, so it is very rarely a problem). 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

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

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:

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

[1] The formula $x = x_0 e^{\lambda t}$ is the solution of the *differential equation* $dx/dt = \lambda x$, and this formulation may be more familiar to some readers. As mentioned at the start of the chapter, differential equations and difference equations are closely related, and these relations are discussed in more detail in Chapter 2.

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)
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 ordinary differential equation (ODE), and we will consider this formulation in the next chapter. 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.

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

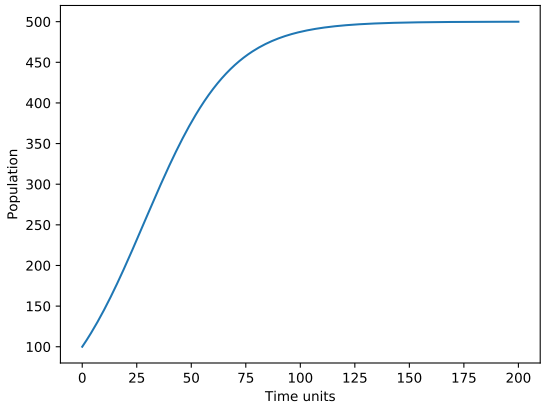

Fig. 1.1 Solution of the logistic growth model for $x_0 = 100, \rho = 5.0, R = 500$.

$$n! = n(n-1)(n-2)\cdots 1, \quad 0! = 1 \tag{1.1}$$

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 (1.1) 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 (1.1) 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. Earlier in the course we introduced Newton's 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 program implementing Newton's method 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 both 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.

1.4 Taylor Series and Approximations

One extremely important use of sequences and series is for approximating other functions. For instance, commonly used functions such as $\sin x, \ln x$, or e^x have been defined to have some desired mathematical properties, but we need some kind of algorithm to evaluate the function values. A convenient approach is to approximate $\sin x$, etc. by polynomials, since they are easy to calculate. It turns out that such approximations exist, for example this result by Gregory from 1667:

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

and an even more amazing result discovered by Taylor in 1715:

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

Here, the notation $d^k f(0)/dx^k$ means the k-th derivative of f evaluated at $x = 0$. 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:

$$f(x) \approx \sum_{k=0}^{N} \frac{1}{k!}\left(\frac{d^k f(0)}{dx^k}\right)x^k. \tag{1.2}$$

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.

As an example, consider the Taylor approximation to the exponential function. For this function we have that $d^k e^x/dx^k = e^x$ for all k, and $e^0 = 1$, and inserting this into (1.2) yields we have

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

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. By a shift of variables we can also make the Taylor polynomials accurate around any point $x = a$:

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

Taylor series formulated as a difference equation. 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

$$e_n = e_{n-1} + \frac{x^{n-1}}{(n-1)!}, \quad e_0 = 0. \tag{1.3}$$

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 (1.3) directly, since we avoid the repeated multiplications involved in the factorial computation.

A complete Python code for solving the difference equation to compute 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 appoximation to be accurate.

Chapter 2
Solving ordinary differential equations

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 Chapter 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.

2.1 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:

$$u'(t) = f(u(t), t), \tag{2.1}$$

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

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

Notice that for generality we write all these right hand sides as functions of both u and t, although the mathematical formulations only involve u. It will become clear later why such a general formulation is useful. Our aim is now to write functions and classes that take f as input, and solve the corresponding ODE to produce u as output.

The Euler method turns an ODE into a difference equation. All the numerical methods we will considered in this chapter are based on approximating the derivatives in the equation $u' = f(u,t)$ by finite differences. This step transforms the ODE into a difference equation, which can be solved with the tools presented in Chapter 1. To introduce the idea, assume that we have computed u at discrete time points $t_0, t_1, \ldots, t_n$. At time t_n we have the ODE

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

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

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

which we may recognize as a difference equation for computing $u(t_{n+1})$ from the known value $u(t_n)$. We can rearrange the terms to obtain an explicit formula for $u(t_{n+1})$:

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

This is known as the *Forward Euler (FE) method*, and is the simplest numerical method for solving and ODE. We can simplify the formula by using the notation for difference equations introduced in Chapter 1. If we let u_n denote the numerical approximation to the exact solution $u(t)$ at $t = t_n$, the difference equation can be written as

$$u_{n+1} = u_n + \Delta t f(u_n, t_n). \tag{2.2}$$

This is a regular difference equation which can be solved using arrays and a for-loop, just as we did for the other difference equations in Chapter 1. We start from the known initial condition u_0, and apply the formula repeatedly to compute u_1, u_2, u_3 and so forth.

An ODE needs an initial condition. In mathematics, an initial condition for u must be specified to have a unique solution of equation (2.1). When solving the equation numerically, we need to set u_0 in order to start our method and compute a solution at all. As an example, consider the very simple ODE

$$u' = u.$$

This equation has the general solution $u = Ce^t$ for any constant C, so it has an infinite number of solutions. Specifying an initial condition $u(0) = u_0$ gives $C = u_0$, and we get the unique solution $u = u_0 e^t$. When solving the equation numerically, we start from our known u_0, and apply formula (2.2) repeatedly:

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

Just as for the difference equations solved in the previous chapter, this repeated application of the same formula is conveniently implemented in a for-loop. For a given time step Δt (`dt`) and number of time steps n, we perform the following steps:

1. Create arrays `t` and `u` of length $n+1$
2. Set initial condition: `u[0]` $= U_0$, `t[0]=0`
3. For $k = 0, 1, 2, \ldots, n-1$:
   - `t[n+1] = t[n] + dt`
   - `u[n+1] = (1 + dt)*u[n]`

A complete Python implementation of this algorithm may look like

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

dt = 0.2
U0 = 1
T = 4
N = int(T/dt)

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

t[0] = 0
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()
```

The solution is shown in Figure 2.1, 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.

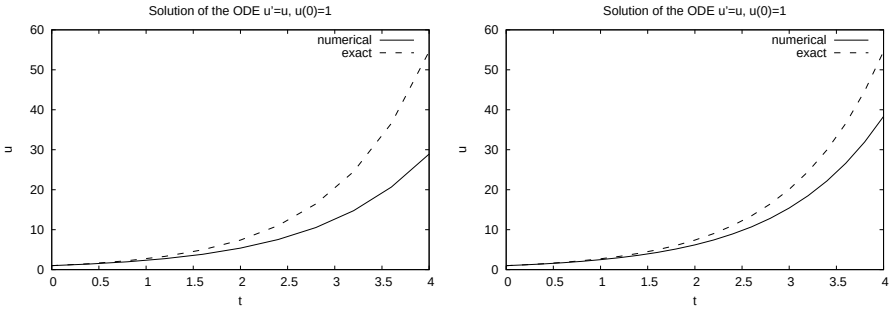

Fig. 2.1 Solution of $u' = u, u(0) = 1$ with $\Delta t = 0.4$ and $\Delta t = 0.2$.

Extending the solver to a 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 $u' = f(u,t)$. This requires a very small modification of the algorithm above;

1. Create arrays `t` and `u` of length $n+1$
2. Set initial condition: `u[0]` $= u_0$, `t[0]=0`
3. For $k = 0, 1, 2, \ldots, n-1$:
   - `u[n+1] = u[n] + dt*f(u[n], t[n])`
   - `t[n+1] = t[n] + dt`

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

```
def ForwardEuler(f, U0, T, N):
    """Solve u'=f(u,t), u(0)=U0, with n steps until t=T."""
    import numpy as np
    t = np.zeros(N+1)
    u = np.zeros(N+1)  # u[n] is the solution at time t[n]

    u[0] = U0
    t[0] = 0
    dt = T/N

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

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

    return u, t
```

This simple function can solve any ODE written on the form (2.1). The right hand side function $f(u,t)$ needs to be implemented as a Python function, and then passed as an argument to `ForwardEuler` together with the initial condition, the stop time `T` and the number of time steps. The two latter arguments are then used to calculate the time step `dt` inside the function.

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 `ForwardEuler` function to solve this problem:

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

U0 = 1
T = 3
N = 30
u, t = ForwardEuler(f, U0, T, N)
plt.plt(t,u)
plt.show()
```

The `ForwardEuler` 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 `u` and `t`, 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 $u' = f(u,t)$, and the function is therefore called as `f(u[n], t[n])` inside `ForwardEuler`. If our right hand side function was defined as `def f(u):` we would get an error message when the function was called inside `ForwardEuler`. This problem is solved by simply writing `def f(u,t):` even if `t` is never used inside the function.[1]

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

1. Identify $f(u,t)$ in your ODE
2. Make sure you have an initial condition u_0
3. Implement the $f(u,t)$ formula in a Python function `f(u, t)`
4. Choose the number of time steps `N`
5. Call `u, t = ForwardEuler(f, U0, T, N)`

[1] 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 `u` and `t`, but, annoyingly, the order of the two arguments varies between different solver libraries. Some expect the `t` argument first, while others expect `u` first.

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 look at a few examples of such methods later in this chapter. However, the FE method is quite suitable for solving most 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.

2.2 The ODE solver implemented as a class

We can increase the flexibility of the `ForwardEuler` solver function by implementing it as a class. The usage of the class may for instance be as follows:

```
method = ForwardEuler_v1(f, U0=0.0, T=40, N=400)
u, t = method.solve()
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 later. For now, let us just look at how such a solver class can be implemented:

- We need a constructor (`__init__`) which takes f, `T`, `N`, and `U0` as arguments

and stores them as attributes.

- The time step Δt and the sequences u_n, t_n must be

initalized and stored as attributes. These tasks are also natural to handle in the constructor.

- The class needs a `solve`-method, which implements the for-loop and returns

the solution, similar to the `ForwardEuler` function considered above.

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_v1:
    def __init__(self, f, U0, T, N):
        self.f, self.U0, self.T, self.N = f, U0, T, N
        self.dt = T/float(N)
```

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

    def solve(self):
        """Compute solution for 0 <= t <= T."""
        self.u[0] = float(self.U0)
        self.t[0] = float(0)

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

    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

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

This class does essentially the same tasks as the `ForwardEuler` function above. 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 the other code can be left unchanged.

We can also use a class to hold the right-hand side $f(u,t)$, 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 the ODE version of the difference equation considered in Chapter 1. The right hand side function has two parameters α and R, but if we want to solve it using our `ForwardEuler` function or class, it must be implemented as a function of u and t only. As we have discussed earlier in the course, a class with a call method provides a very flexible implementation of such a function, since we can set the parameters as attributes in the constructor and use them inside the `__call__` method:

```
class Logistic:
    def __init__(self, alpha, R, U0):
        self.alpha, self.R, self.U0 = alpha, float(R), U0

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

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

```
problem = Logistic(0.2, 1, 0.1)
method = ForwardEuler_v1(problem,problem.U0,40,401)
u, t = method.solve()
plt.plot(t,u)
plt.show()
```

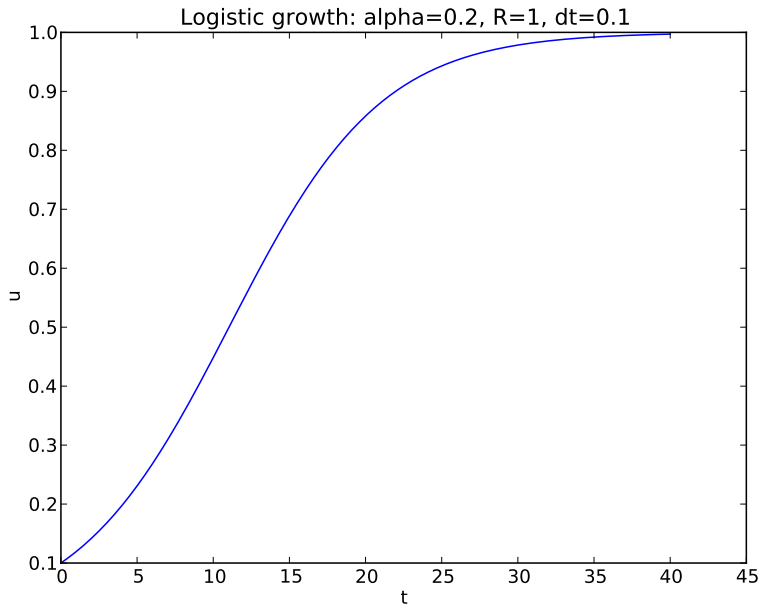

Fig. 2.2 Solution of the logistic growth model.

An alternative class implementation of the FE method. As always in programming, there are multiple ways to perform the same task, and the `ForwardEuler_v1` class presented above is by no means the only possible class implementation of the FE method. A possible alternative implementation is

```
import numpy as np

class ForwardEuler_v2:
    def __init__(self, f):
        self.f = f

    def set_initial_condition(self,U0):
        self.U0 = float(U0)

    def solve(self, time_points):
        """Compute solution for array of time points"""
        self.t = np.asarray(time_points)
```

```
        N = len(self.t)
        self.u = np.zeros(N)
        self.u[0] = self.U0

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

    def advance(self):
        """Advance the solution one time step."""
        # Create local variables to get rid of "self." in
        # the numerical formula
        u, f, n, t = self.u, self.f, self.n, self.t
        #dt is not necessarily constant:
        dt = t[n+1]-t[n]
        unew = u[n] + dt*f(u[n], t[n])
        return unew
```

This class is quite similar to the one above, but we have simplified the constructor considerably, introduced a separate method for setting the initial condition, and changed the `solve` method to take an array of time points as argument. The latter gives a bit more flexibility than the version in `ForwardEuler_v1`, where the stop time and number of time points were passed as arguments to the constructor and used to compute a (constant) time step `dt`. The `ForwardEuler_v2` version does not require the time step to be constant, and the method will work fine if we pass it a `time_points` array with varying distance between the time points. This can be useful if we know that the solution varies rapidly in some time intervals and more slowly in others. However, in most cases we will use an evenly spaced array for the `time_points` argument, for instance created using NumPy's `linspace`, and in such cases there is not much difference between the two FE classes. To consider a concrete example, the solution of the same logistic growth problem as above using the new class may look as follows:

```
problem = Logistic(0.2, 1, 0.1)
time = np.linspace(0,40,401)

method = ForwardEuler_v2(problem)
method.set_initial_condition(problem.U0)
u, t = method.solve(time)
```

2.3 Alternative ODE solvers

As mentioned above, the FE method is not the most sophisticated ODE solver, although it is sufficiently accurate for most of the applications we

will consider here. Many alternative methods exist, with better accuracy and stability than FE. One very popular class of ODE solvers is known as Runge-Kutta methods. The simplest example of a Runge-Kutta method is in fact the FE method;

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

which is an example of a one-stage, first-order, explicit Runge-Kutta method. Being a first-order method, the error in the approximate solution produced by FE is proportional to Δt. We can easily improve the accuracy to second order, i.e., error proportional to Δt^2, by taking one additional step:

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

This method is known as the explicit midpoint method or the modified Euler method. We see that we first compute an intermediate value, often referred to as a *stage derivative*, which is an approximation of the derivative at time t_n. Then, instead of using this value to advance the solution to the next step, which we did in the FE method, we use it to approximate the solution at time $t_n + \Delta t/2$, evaluate the right hand side function again at this midpoint, and use the result (k_2) 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:

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

This method first approximates the derivative at t_n and $t_{n=1}$, and then advances the solution using the average of these two.

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

$$k_i = f(t_n + c_i \Delta t, y_n + \Delta t \sum_{j=1}^{s} a_{ij} k_j), \text{ for } i = 1, \ldots, s \tag{2.3}$$

$$u_{n+1} = u_0 + \Delta t \sum_{i=1}^{s} b_i k_i. \tag{2.4}$$

Here c_i, b_i, a_{ij}, for $i, j, = 1, \ldots, s$ are method-specific, given coefficients. All RK 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. In the ODE literature one often sees these coefficients specified in the form of a *Butcher tableau*, which offers a compact definition of any RK method. The Butcher tableau is simply a specification of all the method coefficients, and for a general RK 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 (2.3)-(2.4) and verify that the you arrive at the correct formulae for the three methods. As an example of a method of higher order, we may consider the the fourth order, four-stage method defined by the Butcher tableau

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

which gives the formulas

$$\begin{aligned} k_1 &= f(u_n, t_n), \\ k_2 &= f(u_n + \frac{1}{2}k_1, t_n + \frac{1}{2}\Delta t), \\ k_3 &= f(u_n + \frac{1}{2}k_2, t_n + \frac{1}{2}\Delta t), \\ k_4 &= f(u_n + k_3, t_n + \Delta t), u_{n+1} = u_n + \frac{\Delta t}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right). \end{aligned}$$

All the RK methods we will consider in this course are explicit methods, which means that $a_{ij} = 0$ for $j \geq i$. If we examine the formula in (2.3), we see that the expression for computing each stage derivative k_i then only includes previously computed stage derivatives, and they can be computed

sequentially from explicit formulas. For implicit RK methods, on the other hand, we we have $a_{ij} \neq 0$ for some $j \geq i$, and we see in (2.3) that the formula for computing k_i will then include k_i on the right hand side. We therefore need to solve equations to compute the stage derivatives, and for non-linear ODEs these will be non-linear equations that are typically solved using Newton's method. This makes implicit RK 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 not consider implicit RK methods in this course.

2.4 A class hierarchy of ODE solvers

We now want to implement some of the Runge-Kutta methods as classes, similar to the FE classes introduced above. When inspecting the `ForwardEuler_v2` 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 very convenient for implementing such a collection of ODE solvers. In this way we can collect all common code in a superclass, 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 corresponding time levels t_n, $k = 0, 1, 2, \ldots, n$
- Storing the right-hand side function $f(u, t)$
- Storing and applying 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 sub-classes. 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 may look like

```
import numpy as np

class ODESolver:
    def __init__(self, f):
        self.f = f

    def advance(self):
        """Advance solution one time step."""
        raise NotImplementedError  # implement in subclass
```

```
    def set_initial_condition(self, U0):
        self.U0 = float(U0)

    def solve(self, time_points):
        self.t = np.asarray(time_points)
        N = len(self.t)
        self.u = np.zeros(N)
        # Assume that self.t[0] corresponds to self.U0
        self.u[0] = self.U0

        # Time loop
        for n in range(N-1):
            self.n = n
            self.u[n+1] = self.advance()
        return self.u, self.t
```

Notice that the `ODESolver` is meant to be a pure superclass, and we have therefore implemented the `advance`-method to raise an exception if the class is used on its own. We could also have omitted the `advance`-method from the superclass altogether, but the chosen implementation makes it clearer to users of the class that `ODESolver` is a pure superclass.

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

```
class ForwardEuler(ODESolver):
    def advance(self):
        u, f, n, t = self.u, self.f, self.n, self.t

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

Similarly, the explicit midpoint method and the fourth-order RK method can be subclasses, each with a single method:

```
class ExplicitMidpoint(ODESolver):
    def advance(self):
        u, f, n, t = self.u, self.f, self.n, self.t
        dt = t[n+1] - t[n]
        dt2 = dt/2.0
        k1 = f(u[n], t)
        k2 = f(u[n] + dt2*k1, t[n] + dt2)
        unew = u[n] + dt*k2
        return unew

class RungeKutta4(ODESolver):
    def advance(self):
        u, f, n, t = self.u, self.f, self.n, self.t
        dt = t[n+1] - t[n]
        dt2 = dt/2.0
        k1 = f(u[n], t)
        k2 = f(u[n] + dt2*k1, t[n] + dt2)
```

```
        k3 = f(u[n] + dt2*k2, t[n] + dt2)
        k4 = f(u[n] + dt*k3, t[n] + dt)
        unew = u[n] + (dt/6.0)*(k1 + 2*k2 + 2*k3 + k4)
        return unew
```

The use of these classes is nearly identical to the second version of the FE class introduced earlier. Considering the same simple ODE used above; $u' = u$, $u(0) = 1$, $t \in [0,3]$, $\Delta t = 0.1$, the code looks like:

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

def f(u, t):
    return u

time_points = np.linspace(0, 3, 11)

fe = ForwardEuler(f)
fe.set_initial_condition(U0=1)
u1, t1 = fe.solve(time_points)
plt.plot(t1, u1, label='Forward Euler')

em = ExplicitMidpoint(f)
em.set_initial_condition(U0=1)
u2, t2 = em.solve(time_points)
plt.plot(t2, u2, label='Explicit Midpoint')

rk4 = RungeKutta4(f)
rk4.set_initial_condition(U0=1)
u3, t3 = rk4.solve(time_points)
plt.plot(t3, u3, label='RungeKutta 4')

#plot the exact solution in the same plot
time_exact = np.linspace(0,3,301) #more points to improve the plot
plt.plot(time_exact,np.exp(time_exact),label='Exact')

plt.legend()
plt.show()
```

This code will solve the same equation using three different methods, and plot the solutions in the same window. Experimenting with different step sizes should reveal the difference in accuracy between the two methods.

Chapter 3
Solving 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

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

The solution of this system is $u = \cos t, v = \sin t$ which can easily be verified by insterting the solution into the equations and 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 chapter we will extend the solvers introduced in Chapter 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 general mathematical notation. We have n unknowns

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

in a system of n ODEs:

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

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

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

and

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

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

$$u' = f(u,t), \quad u(0) = u_0$$

where u and f are 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.

3.1 An `ODESolver` class for systems of ODEs

The `ODESolver` class above was written for a scalar ODE. 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 start with the simplest method. 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

$$u^{(i)}_{k+1} = u^{(i)}_k + \Delta t f^{(i)}(u_k, t_k), \text{ for } i = 0, \ldots, n-1,$$

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 idential to the version for scalar ODEs;

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

with the important difference that both u[k] and u[k+1] are now arrays.[1] 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 first 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:

```
t = [0.  0.4 0.8 1.2 (...) ]

u = [ 1.0 1.4  1.96 2.744  (...)]

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. Indexing into it yields one-dimensional arrays of length two, which are the two solution components at each time step:

```
u = [[1.0 0.8][1.4 1.1] [1.9 2.7] (...)]

u[0] = [1.0 0.8]
u[1] = [1.4 1.1]

(...)
```

To make a generic ODESolver class that works both with scalars and systems, we need to make the following changes to the class from the previous chapter:

- Ensure that f(u,t) always returns an array.
- Inspect U_0 to see if it is a single number or a list/array/tuple and make the corresponding u a one-dimensional or two-dimensional array

If these two items are handled and initialized correctly, the rest of the code from Chapter 2 will work with no modifications. The extended superclass implementation may look like:

```
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)
```

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

```
        self.f = lambda u, t: np.asarray(f(u, t), float)

    def set_initial_condition(self, U0):
        if isinstance(U0, (float,int)):  # 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, time_points):
        self.t = np.asarray(time_points)
        N = len(self.t)
        if self.neq == 1:  # scalar ODEs
            self.u = np.zeros(N)
        else:              # systems of ODEs
            self.u = np.zeros((N,self.neq))

        # Assume that self.t[0] corresponds to self.U0
        self.u[0] = self.U0

        # Time loop
        for n in range(N-1):
            self.n = n
            self.u[n+1] = self.advance()
        return self.u, self.t
```

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 `asarray` to convert any `f` that returns a list or tuple to a function returning a NumPy array. This modification is not strictly needed, since we could just assume that the user implements `f` so that it returns an array, but it makes the class more robust and flexible. Similar tests are included in the `set_initial_condition`, to make sure that `U0` is either a single number (`float`) or a NumPy array, and to set 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, as well the implementation of the `advance` method in the subclasses, can be left unchanged.

Example: ODE model for throwing a ball. To demonstrate the use of the extended `ODESolver` hierarchy, let us consider a system of ODEs describing the trajectory of a ball. We define $x(t), y(t)$ to be the position of the ball, v_x and v_y the velocity components, and a_x, a_y the acceleration components. From the definition of velocity and acceleration, we have $v_x = dx/dt, v_y = dy/dt, a_x = dv_x/dt$, and $a_y = dv_y/dt$. If we neglect air resistance there are no forces acting on the ball in the x-direction, so from Newton's second law $a_x = 0$, while the acceleration in the y-direction must be equal to the acceleration of gravity. We get

$$a_x = 0 \quad \Rightarrow \quad \frac{dv_x}{dt} = 0,$$
$$a_y = -g \quad \Rightarrow \quad \frac{dv_y}{dt} = -g\ .$$

We can easily add air resistance or other additional physics to these equations later if needed. The complete ODE system can be written as

$$\frac{dx}{dt} = v_x,$$
$$\frac{dv_x}{dt} = 0,$$
$$\frac{dy}{dt} = v_y,$$
$$\frac{dv_y}{dt} = -g,$$

and solve the system we need to define initial conditions for all these variables, i.e., we need to know the initial position and velocity of the ball. We want to use the ODESolver class hierarchy to solve the system, and first need to implement the right hand side as a Python function:

```
def f(u, t):
    x, vx, y, vy = u
    g = 9.81
    return [vx, 0, vy, -g]
```

We see that the function here returns a list, but this will automatically be converted to 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 four components:

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

# Initial condition, start at the origin:
x = 0; y = 0
# velocity magnitude and angle:
v0 = 5; theta = 80*np.pi/180
vx = v0*np.cos(theta); vy = v0*np.sin(theta)

U0 = [x, vx, y, vy]

solver= ForwardEuler(f)
solver.set_initial_condition(U0)
time_points = np.linspace(0, 1.0, 101)
u, t = solver.solve(time_points)
# u is an array of [x,vx,y,vy] arrays, plot y vs x:
x = u[:,0];  y = u[:,2]
```

```
plt.plot(x, y)
plt.show()
```

Notice that since `u` is a two-dimensional array, we use array slicing to extract and plot the individual components. A call like `plt.plot(t,u)` will also work, but it will plot all the solution components in the same window, which is not very useful in this case.

Chapter 4
Modeling infectious diseases

In this chapter we will look at a particular family of ODE systems that describe the spread of infectious diseases. Athough the spread of infections is a very complex physical and biological process, we shall see that it can be modeled with fairly simple systems of ODEs, which we can solve using the tools from the previous chapters.

4.1 Derivation of the SIR model

In order to derive a model we need to make a number of simplifying assumptions. The most important one is that we do not consider individuals, just populations. 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 three categories of people:

- **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, consider the dynamics in a time interval Δt. During this time, a certain number of infected people will meet and transfer the infection to susceptible people. It is natural that such disease transmission

is proportional to the product SI, since in a mix of S susceptibles and I infected people, there are SI possible pairs containing one person in the I category and one in S. We now assume that over the time interval Δt, a certain fraction of these possible S-I encounters actually happen and lead to disease transmission, and we introduce a constant β to represent this fraction. We can interpret β as the probability that an infected person meets and infects a susceptible person. Over the time interval Δt, we then have $\Delta t\,\beta SI$ people who get infected and move from the S to the I category. With mathematics, we have the loss in $S(t)$ from time t to $t+\Delta t$ given by

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

and a corresponding gain in $I(t)$:

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

These two equations represent the key component of all the models considered in this chapter. 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. 4.1 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 catogory. 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:

$$S(t+\Delta t) = S(t) - \Delta t\,\beta S(t)I(t) \tag{4.1}$$
$$I(t+\Delta t) = I(t) + \Delta t\,\beta S(t)I(t) - \Delta t\nu I(t) \tag{4.2}$$
$$R(t+\Delta t) = R(t) + \Delta t\,\nu I(t) \tag{4.3}$$

Although the notation is slightly different, we may recognize these equations as a system of difference equations of the same kind that we solved in Chapter 1. We could easily solve the equations as such, using techniques from Chapter 1, but models of this kind are more commonly formulated as systems of ODEs, which can be solved with the tools we developed in Chapter 3.

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

$$\frac{S(t+\Delta t) - S(t)}{\Delta t} = -\beta S(t)I(t), \tag{4.4}$$
$$\frac{I(t+\Delta t) - I(t)}{\Delta t} = \beta t S(t)I(t) - \nu I(t), \tag{4.5}$$
$$\frac{R(t+\Delta t) - R(t)}{\Delta t} = \nu I(t). \tag{4.6}$$

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

$$S'(t) = -\beta SI \tag{4.7}$$
$$I'(t) = \beta SI - \nu I \tag{4.8}$$
$$R'(t) = \nu I \tag{4.9}$$

This 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 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.

Although the system (4.7)-(4.9) looks quite simple, it is not possible to solve these equations analytically. For particular applications it is common to make simplifications that allow analytical solutions. For instance, when studying the early phase of an epidemic one is mostly interested in the I 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. Setting S to be constant turns (4.8) into a simple equation describing expo-

nential growth, with solution

$$I(t) = I_0 e^{(\beta S - \nu)}. \tag{4.10}$$

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 (4.10) fits the observed dynamics. However, if we want to describe the full dynamics of the epidemic we need to solve the complete system of ODEs, and in this case there are no analytical solutions available.

Solving the SIR model with the `ODESystem` class hierarchy. We could of course implement a numerical solution of the SIR equations directly, for instance by applying the forward Euler method to (4.7)-(4.9), which will simply give us back the original difference equations in (4.4)-(4.6). However, since the ODE solver tools we developed in Chapter 3 are completely general, they can easily be used to solve the SIR model. To solve the system using the fourth-order RK method of the ODESolver hierarchy, the Python implementation may look as follows:

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

def SIR_model(u,t):
    beta = 0.001
    nu = 1/7.0
    S, I, R = u[0], u[1], u[2]
    dS = -beta*S*I
    dI = beta*S*I - 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, 101)
u, t = solver.solve(time_points)
S = u[:,0];  I = u[:,1]; R = u[:,2]

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

A class implementation of the SIR model. As noted above, estimating the parameters in the model is often challenging. In fact, the most important application of models of this kind is to predict the dynamics of new diseases, for instance 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. Although there are ways to estimate the parameters from the early disease dynamics, the estimates will contain a large degree of uncertainty, and a common strategy is then to run the model for multiple parameters to get an idea of what disease outbreak scenarios to expect. 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]
        dS = -self.beta*S*I
        dI = self.beta*S*I - self.nu*I
        dR = self.nu*I
        return [dS,dI,dR]
```

The use of the class is very similar to the use of the `SIR_model` function above. We need to 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.

4.2 Extending the SIR model

The SIR model itself 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.

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 (4.7)-(4.9) 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

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

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.

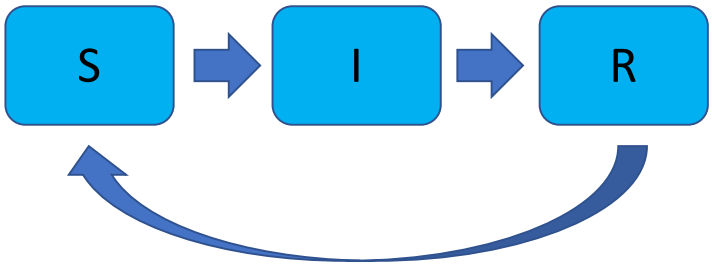

Fig. 4.2 Illustration of a SIR model without lifelong immunity, where people move from the R category back to S after a given time.

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 an additional 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 4.3, and the ODEs may look like

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

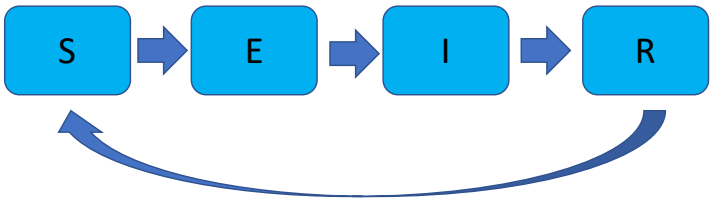

Fig. 4.3 Illustration of the SEIR model, without life-long immunity.

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
        dS = -self.beta*S*I + self.gamma*R
        dE  = self.beta*S*I - 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=0.001, 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()
```

4.3 A model of the Covid19 pandemic

The models considered above can typically 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 has been used by Norwegian health authorities to predict the spread of the 2020 Covid19 pandemic. 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.[1] A stochastic model is somewhat more flexible than the deterministic ODE version, and 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.

To describe Covid19, the SEIR model introduced above is modified to incorporate two important disease characteristics:

- A certain number of people infected with Covid19 have no symptoms. 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.
- A large number of infections occur before the infector experiences symptoms, which suggests an additional *exposed* category where people are infectious but do not yet experience symptoms.

These characteristics can be modeled by adding more categories to the SEIR model introduced earlier. We include two exposed categories 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 4.4.

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

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

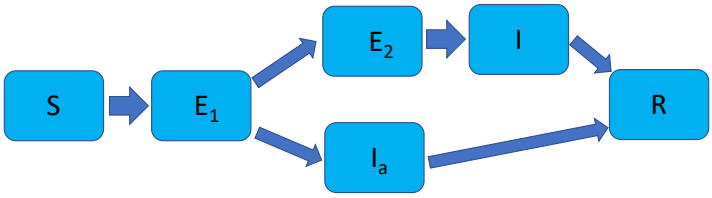

Fig. 4.4 Illustration of the Covid19 epidemic model, with two alternative disease trajectories.

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: $\beta SI\Delta t$.
- Infected by people in I_a: $r_{ia}\beta SI_a\Delta t$
- Infected by people in E_2: $r_{e2}\beta SE_2\Delta t$

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 a ODE, we get the following equation for the S category:

$$\frac{dS}{dt} = -\beta SI - r_{ia}\beta SI_a - r_{e2}\beta SE_2. \tag{4.11}$$

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

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

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.

The E_2 category 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

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

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 probably not correct for Covid19, but since the duration of immunity is still largely unknown, and we are mostly interested in the early epidemic spread, we neglect the loss of immunity.

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

$$\begin{aligned}
S'(t) &= -\beta SI - r_{ia}\beta SI_a - r_{e2}\beta SE_2,\\
E_1'(t) &= \beta SI + r_{ia}\beta SI_a + r_{e2}\beta SE_2 - \lambda_1 E_1,\\
E_2'(t) &= \lambda_1(1-p_a)E_1 - \lambda_2 E_2,\\
I'(t) &= \lambda_2 E_2 - \mu I,\\
I_a'(t) &= \lambda_1 p_a E_1 - \mu I_a,\\
R'(t) &= \mu(I + I_a).
\end{aligned}$$

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

Parameter	Value
β	$6.0 \cdot 10^{-6}$
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 based on the early phase of the Covid19 outbreak and can tell us quite a bit about the disease dynamics. 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, while the mean duration of the infectious incubation

period (E_2) is two days. Furthermore, we see that the mean infectiousness of asymptomatic people is 10% of the infectiousness of the symptomatic cases. However, the infectiousness of the E_2 category is 25% higher than the infectiousness of the I category. This increased infectiousness is most likely the result of the E_2 category being asymptomatic, so these people will move around a lot more than the symptomatic I category. The I_a group is also, of course, asymptomatic and therefore likely to move around more, but it is assumed that these people have a very low virus count and are therefore less infectious than the people that develop symptoms.

A function implementation of the SEEIIR model can look as follows

```
def SEEIIR_model(u,t):
    beta = 6.0e-6; r_ia = 0.1; r_e2=1.25;
    lmbda1=0.33; lmbda2=0.5; p_a=0.4; mu=0.2;

    S, E1, E2, I, Ia, R = u

    dS  = -beta*S*I - r_ia*beta*S*Ia - r_e2*beta*S*E2
    dE1 = beta*S*I + r_ia*beta*S*Ia + r_e2*beta*S*E2 - lmbda_1*E1
    dE2 = lmbda_1*(1-p_a)*E1 - lmbda_2*E2
    dI  = lambda_2*E2 - mu*I
    dIa = lambda_1*p_a*E1 - mu*Ia
    dR  = mu*(I + Ia)
    return [dS, dE1, dE2, dI, dIa, dR]
```

Just as the simpler models, the SEEIIR model can be solved with methods in the `ODEsolver` class hierarchy:

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

S_0 = 5e6
E1_0 = 0
E2_0 = 100
I_0 = 0
Ia_0 = 0
R_0 = 0
U0 = [S0, E1_0, E2_0, I_0, Ia_0, R_0]

model = SEEIIR(beta=0.001, mu=1.0/5,nu=1.0/7,gamma=1.0/50)

solver = RungeKutta4(model)
solver.set_initial_condition(U0)
time_points = np.linspace(0, 100, 101)
u, t = solver.solve(time_points)
S = u[:,0]; E1 = u[:,1]; E2 = u[:,2];
I = u[:,3]; Ia = u[:,4]; R = u[:,5]

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