

This chapter introduces the basic ideas of object-oriented programming. Different people put different meanings into the term object-oriented programming: Some use the term for programming with objects in general, while others use the term for programming with class hierarchies. The author applies the second meaning, which is the most widely accepted one in computer science. The first meaning is better named *object-based* programming. Since everything in Python is an object, we do object-based programming all the time, yet one usually reserves this term for the case when classes different from Python's basic types (`int`, `float`, `str`, `list`, `tuple`, `dict`) are involved.

A necessary background for the present chapter is Chapter 7. For Chapters 9.2 and 9.3 one must know basic methods for numerical differentiation and integration, for example from Appendix A. Chapter 9.4 requires knowledge of numerical solution of ordinary differential equations, which is treated in Appendix B and Chapter 7.4. It takes time to grasp the ideas of object-oriented programming, but it will hopefully become clear through many examples. During the initial readings of the chapter, it can be beneficial to skip the more advanced material in Chapters 9.2.3–9.2.6 and 9.4.7–9.4.9.

All the programs associated with this chapter are found in the `src/oo` folder.

## 9.1 Inheritance and Class Hierarchies

Most of this chapter tells you how to put related classes together in families such that the family can be viewed as one unit. This idea helps to hide details in a program, and makes it easier to modify or extend the program.

A family of classes is known as a *class hierarchy*. As in a biological family, there are parent classes and child classes. Child classes can *inherit* data and methods from parent classes, they can modify these data and methods, and they can add their own data and methods. This means that if we have a class with some functionality, we can extend this class by creating a child class and simply add the functionality we need. The original class is still available and the separate child class is small, since it does not need to repeat the code in the parent class.

The magic of object-oriented programming is that other parts of the code do not need to distinguish whether an object is the parent or the child – all generations in a family tree can be treated as a unified object. In other words, one piece of code can work with all members in a class family or hierarchy. This principle has revolutionized the development of large computer systems<sup>1</sup>.

The concepts of classes and object-oriented programming first appeared in the Simula programming language in the 1960s. Simula was invented by the Norwegian computer scientists Ole-Johan Dahl and Kristen Nygaard, and the impact of the language is particularly evident in C++, Java, and C#, three of the most dominating programming languages in the world today. The invention of object-oriented programming was a remarkable achievement, and the professors Dahl and Nygaard received two very prestigious prizes: the von Neumann medal and the Turing prize (popularly known as the Nobel prize of computer science).

A parent class is usually called *base class* or *superclass*, while the child class is known as a *subclass* or *derived class*. We shall use the terms superclass and subclass from now on.

### 9.1.1 A Class for Straight Lines

Assume that we have written a class for straight lines,  $y = c_0 + c_1x$ :

```
class Line:
    def __init__(self, c0, c1):
        self.c0 = c0
        self.c1 = c1

    def __call__(self, x):
        return self.c0 + self.c1*x

    def table(self, L, R, n):
        """Return a table with n points for L <= x <= R."""
        s = ''
        for x in linspace(L, R, n):
            y = self(x)
            s += '%12g %12g\n' % (x, y)
        return s
```

<sup>1</sup> Two of the most widely used computer languages today are Java and C#. Both of them force programs to be written in an object-oriented style.

The constructor `__init__` initializes the coefficients  $c_0$  and  $c_1$  in the expression for the straight line:  $y = c_0 + c_1x$ . The call operator `__call__` evaluates the function  $c_1x + c_0$ , while the `table` method samples the function at  $n$  points and creates a table of  $x$  and  $y$  values.

### 9.1.2 A First Try on a Class for Parabolas

A parabola  $y = c_0 + c_1x + c_2x^2$  contains a straight line as a special case ( $c_2 = 0$ ). A class for parabolas will therefore be similar to a class for straight lines. All we have to do is to add the new term  $c_2x^2$  in the function evaluation and store  $c_2$  in the constructor:

```
class Parabola:
    def __init__(self, c0, c1, c2):
        self.c0 = c0
        self.c1 = c1
        self.c2 = c2

    def __call__(self, x):
        return self.c2*x**2 + self.c1*x + self.c0

    def table(self, L, R, n):
        """Return a table with n points for L <= x <= R."""
        s = ''
        for x in linspace(L, R, n):
            y = self(x)
            s += '%12g %12g\n' % (x, y)
        return s
```

Observe that we can copy the `table` method from class `Line` without any modifications.

### 9.1.3 A Class for Parabolas Using Inheritance

Python and other languages that support object-oriented programming have a special construct, so that class `Parabola` does not need to repeat the code that we have already written in class `Line`. We can specify that class `Parabola` *inherits* all code from class `Line` by adding “(Line)” in the class headline:

```
class Parabola(Line):
```

Class `Parabola` now automatically gets all the code from class `Line` – invisibly. Exercise 9.1 asks you to explicitly demonstrate the validity of this assertion. We say that class `Parabola` is *derived* from class `Line`, or equivalently, that class `Parabola` is a subclass of its superclass `Line`.

Now, class `Parabola` should not be identical to class `Line`: it needs to add data in the constructor (for the new term) and to modify the call operator (because of the new term), but the `table` method can be inherited as it is. If we implement the constructor and the call operator

in class `Parabola`, these will *override* the inherited versions from class `Line`. If we do not implement a `table` method, the one inherited from class `Line` is available as if it were coded visibly in class `Parabola`.

Class `Parabola` must first have the statements from the class `Line` methods `__call__` and `__init__`, and then add extra code in these methods. An important principle in computer programming is to avoid repeating code. We should therefore call up functionality in class `Line` instead of copying statements from class `Line` methods to `Parabola` methods. Any method in the superclass `Line` can be called using the syntax

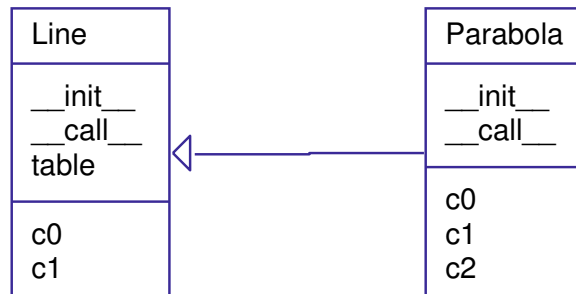
```
Line.methodname(self, arg1, arg2, ...)
# or
super(Line, self).methodname(arg1, arg2, ...)
```

Let us now show how to write class `Parabola` as a subclass of class `Line`, and implement just the new additional code that we need and that is not already written in the superclass:

```
class Parabola(Line):
    def __init__(self, c0, c1, c2):
        Line.__init__(self, c0, c1) # let Line store c0 and c1
        self.c2 = c2

    def __call__(self, x):
        return Line.__call__(self, x) + self.c2*x**2
```

This short implementation of class `Parabola` provides exactly the same functionality as the first version of class `Parabola` that we showed on page 481 and that did not inherit from class `Line`. Figure 9.1 shows the class hierarchy in UML fashion. The arrow from one class to another indicates inheritance.



**Fig. 9.1** UML diagram for the class hierarchy with superclass `Line` and subclass `Parabola`.

A quick demo of the `Parabola` class in a main program,

```
p = Parabola(1, -2, 2)
p1 = p(x=2.5)
print p1
print p.table(0, 1, 3)
```

gives this output:

8.5		
	0	1
	0.5	0.5
	1	1

*Program Flow.* The program flow can be somewhat complicated when we work with class hierarchies. Consider the code segment

```
p = Parabola(1, -1, 2)
p1 = p(x=2.5)
```

Let us explain the program flow in detail for these two statements. As always, you can monitor the program flow in a debugger as explained in Chapter D.1.

Calling `Parabola(1, -1, 2)` leads to a call to the constructor method `__init__`, where the arguments `c0`, `c1`, and `c2` in this case are `int` objects with values 1, -1, and 2. The `self` argument in the constructor is the object that will be returned and referred to by the variable `p`. Inside the constructor in class `Parabola` we call the constructor in class `Line`. In this latter method, we create two attributes in the `self` object. Printing out `dir(self)` will explicitly demonstrate what `self` contains so far in the construction process. Back in class `Parabola`'s constructor, we add a third attribute `c2` to the same `self` object. Then the `self` object is invisibly returned and referred to by `p`.

The other statement, `p1 = p(x=2.5)`, has a similar program flow. First we enter the `p.__call__` method with `self` as `p` and `x` as a `float` object with value 2.5. The program flow jumps to the `__call__` method in class `Line` for evaluating the linear part  $c_1x + c_0$  of the expression for the parabola, and then the flow jumps back to the `__call__` method in class `Parabola` where we add the new quadratic term.

#### 9.1.4 Checking the Class Type

Python has the function `isinstance(i,t)` for checking if an instance `i` is of class type `t`:

```
>>> l = Line(-1, 1)
>>> isinstance(l, Line)
True
>>> isinstance(l, Parabola)
False
```

A `Line` is not a `Parabola`, but is a `Parabola` a `Line`?

```
>>> p = Parabola(-1, 0, 10)
>>> isinstance(p, Parabola)
True
>>> isinstance(p, Line)
True
```

Yes, from a class hierarchy perspective, a `Parabola` instance is regarded as a `Line` instance too, since it contains everything that a `Line` instance contains.

Every instance has an attribute `__class__` that holds the type of class:

```
>>> p.__class__
<class __main__.Parabola at 0xb68f108c>
>>> p.__class__ == Parabola
True
>>> p.__class__.__name__    # string version of the class name
'Parabola'
```

Note that `p.__class__` is a class object (or class definition one may say<sup>2</sup>), while `p.__class__.__name__` is a string. These two variables can be used as an alternative test for the class type:

```
if p.__class__.__name__ == 'Parabola':
    <statements>
# or
if p.__class__ == Parabola:
    <statements>
```

However, `isinstance(p, Parabola)` is the recommended programming style for checking the type of an object.

A function `issubclass(c1, c2)` tests if class `c1` is a subclass of class `c2`, e.g.,

```
>>> issubclass(Parabola, Line)
True
>>> issubclass(Line, Parabola)
False
```

The superclasses of a class are stored as a tuple in the `__bases__` attribute of the class object:

```
>>> p.__class__.__bases__
(<class __main__.Line at 0xb7c5d2fc>,)
>>> p.__class__.__bases__[0].__name__    # extract name as string
'Line'
```

### 9.1.5 Attribute versus Inheritance

Instead of letting class `Parabola` inherit from a class `Line`, we may let it *contain* a class `Line` instance as an attribute:

```
class Parabola:
    def __init__(self, c0, c1, c2):
        self.line = Line(c0, c1)    # let Line store c0 and c1
        self.c2 = c2
```

<sup>2</sup> This means that even the definition of a class, i.e., the class code, is an object that can be referred to by a variable. This is useful in many occasions, see pages 504 and 519.

```
def __call__(self, x):  
    return self.line(x) + self.c2*x**2
```

Whether to use inheritance or an attribute depends on the problem being solved. If it is natural to say that class `Parabola` *is* a `Line` object, we say that `Parabola` has an *is-a relationship* with class `Line`. Alternatively, if it is natural to think that class `Parabola` *has a* `Line` object, we speak about a *has-a relationship* with class `Line`. In the present example, the is-a relationship is most natural since a special case of a parabola is a straight line.

### 9.1.6 Extending versus Restricting Functionality

In our example of `Parabola` as a subclass of `Line`, we used inheritance to *extend* the functionality of the superclass. Inheritance can also be used for *restricting* functionality. Say we have class `Parabola`:

```
class Parabola:  
    def __init__(self, c0, c1, c2):  
        self.c0 = c0  
        self.c1 = c1  
        self.c2 = c2  
  
    def __call__(self, x):  
        return self.c2*x**2 + self.c1*x + self.c0  
  
    def table(self, L, R, n):  
        ...
```

We can define `Line` as a subclass of `Parabola` and restrict the functionality:

```
class Line(Parabola):  
    def __init__(self, c0, c1):  
        Parabola.__init__(self, c0, c1, 0)
```

The `__call__` and `table` methods can be inherited as they are defined in class `Parabola`.

From this example it becomes clear that there is no unique way of arranging classes in hierarchies. Rather than starting with `Line` and introducing `Parabola`, `Cubic`, and perhaps eventually a general `Polynomial` class, we can start with a general `Polynomial` class and let `Parabola` be a subclass which restricts all coefficients except the first three to be zero. Class `Line` can then be a subclass of `Parabola`, restricting the value of one more coefficient. Exercise 9.4 asks you to implement such a class hierarchy, and to discuss what kind of hierarchy design you like best.

### 9.1.7 Superclass for Defining an Interface

As another example of class hierarchies, we now want to represent functions by classes, as described in Chapter 7.1.2, but in addition to the `__call__` method, we also want to provide methods for the first and second derivative. The class can be sketched as

```
class SomeFunc:
    def __init__(self, parameter1, parameter2, ...):
        # store parameters
    def __call__(self, x):
        # evaluate function
    def df(self, x):
        # evaluate the first derivative
    def ddf(self, x):
        # evaluate the second derivative
```

For a given function, the analytical expressions for first and second derivative must be manually coded. However, we could think of inheriting general functions for computing these derivatives numerically, such that the only thing we must always implement is the function itself. To realize this idea, we create a superclass<sup>3</sup>

```
class FuncWithDerivatives:
    def __init__(self, h=1.0E-9):
        self.h = h # spacing for numerical derivatives

    def __call__(self, x):
        raise NotImplementedError\
        ('__call__ missing in class %s' % self.__class__.__name__)

    def df(self, x):
        # compute first derivative by a finite difference:
        h = self.h
        return (self(x+h) - self(x-h))/(2.0*h)

    def ddf(self, x):
        # compute second derivative by a finite difference:
        h = self.h
        return (self(x+h) - 2*self(x) + self(x-h))/(float(h)**2)
```

This class is only meant as a superclass of other classes. For a particular function, say  $f(x) = \cos(ax) + x^3$ , we represent it by a subclass:

```
class MyFunc(FuncWithDerivatives):
    def __init__(self, a):
        self.a = a

    def __call__(self, x):
        return cos(self.a*x) + x**3

    def df(self, x):
        a = self.a
        return -a*sin(a*x) + 3*x**2

    def ddf(self, x):
        a = self.a
        return -a*a*cos(a*x) + 6*x
```

<sup>3</sup> Observe that we carefully ensure that the divisions in methods `df` and `ddf` can never be integer divisions.



The superclass constructor is never called, hence `h` is never initialized, and there are no possibilities for using numerical approximations via the superclass methods `df` and `ddf`. Instead, we override all the inherited methods and implement our own versions. Many think it is a good programming style to always call the superclass constructor in a subclass constructor, even in simple classes where we do not need the functionality of the superclass constructor.

For a more complicated function, e.g.,  $f(x) = \ln |p \tanh(qx \cos rx)|$ , we may skip the analytical derivation of the derivatives, and just code  $f(x)$  and rely on the difference approximations inherited from the superclass to compute the derivatives:

```
class MyComplicatedFunc(FuncWithDerivatives):
    def __init__(self, p, q, r, h=1.0E-9):
        FuncWithDerivatives.__init__(self, h)
        self.p, self.q, self.r = p, q, r

    def __call__(self, x):
        return log(abs(self.p*tanh(self.q*x*cos(self.r*x))))
```

That's it! We are now ready to use this class:

```
>>> f = MyComplicatedFunc(1, 1, 1)
>>> x = pi/2
>>> f(x)
-36.880306514638988
>>> f.df(x)
-60.593693618216086
>>> f.ddf(x)
3.3217246931444789e+19
```

Class `MyComplicatedFunc` inherits the `df` and `ddf` methods from the superclass `FuncWithDerivatives`. These methods compute the first and second derivatives approximately, provided that we have defined a `__call__` method. If we fail to define this method, we will inherit `__call__` from the superclass, which just raises an exception, saying that the method is not properly implemented in class `MyComplicatedFunc`.

The important message in this subsection is that we introduced a super class to mainly define an *interface*, i.e., the operations (in terms of methods) that one can do with a class in this class hierarchy. The superclass itself is of no direct use, since it does not implement any function evaluation in the `__call__` method. However, it stores a variable common to all subclasses (`h`), and it implements general methods `df` and `ddf` that any subclass can make use of. A specific mathematical function must be represented as a subclass, where the programmer can decide whether analytical derivatives are to be used, or if the more lazy approach of inheriting general functionality (`df` and `ddf`) for computing numerical derivatives is satisfactory.

In object-oriented programming, the superclass very often defines an interface, and instances of the superclass have no applications on their own – only instances of subclasses can do anything useful.

To digest the present material on inheritance, we recommend to do Exercises 9.1–9.4 before reading the next section.

## 9.2 Class Hierarchy for Numerical Differentiation

Chapter 7.3.2 presents a class `Derivative` that “can differentiate” any mathematical function represented by a callable Python object. The class employs the simplest possible numerical derivative. There are a lot of other numerical formulas for computing approximations to  $f'(x)$ :

$$f'(x) = \frac{f(x+h) - f(x)}{h} + \mathcal{O}(h), \quad (\text{1st-order forward diff.}) \quad (9.1)$$

$$f'(x) = \frac{f(x) - f(x-h)}{h} + \mathcal{O}(h), \quad (\text{1st-order backward diff.}) \quad (9.2)$$

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + \mathcal{O}(h^2), \quad (\text{2nd-order central diff.}) \quad (9.3)$$

$$f'(x) = \frac{4}{3} \frac{f(x+h) - f(x-h)}{2h} - \frac{1}{3} \frac{f(x+2h) - f(x-2h)}{4h} + \mathcal{O}(h^4),$$

(4th-order central diff.) (9.4)

$$f'(x) = \frac{3}{2} \frac{f(x+h) - f(x-h)}{2h} - \frac{3}{5} \frac{f(x+2h) - f(x-2h)}{4h} +$$

$$\frac{1}{10} \frac{f(x+3h) - f(x-3h)}{6h} + \mathcal{O}(h^6),$$

(6th-order central diff.) (9.5)

$$f'(x) = \frac{1}{h} \left( -\frac{1}{6}f(x+2h) + f(x+h) - \frac{1}{2}f(x) - \frac{1}{3}f(x-h) \right) + \mathcal{O}(h^3),$$

(3rd-order forward diff.) (9.6)

The key ideas about the implementation of such a family of formulas are explained in Chapter 9.2.1. For the interested reader, Chapters 9.2.3–9.2.6 contains more advanced additional material that can well be skipped in a first reading. However, the additional material puts the basic solution in Chapter 9.2.1 into a wider perspective, which may increase the understanding of object orientation.

### 9.2.1 Classes for Differentiation

It is argued in Chapter 7.3.2 that it is wise to implement a numerical differentiation formula as a class where  $f(x)$  and  $h$  are attributes and a `__call__` method makes class instances behave as ordinary Python

functions. Hence, when we have a collection of different numerical differentiation formulas, like (9.1)–(9.6), it makes sense to implement each one of them as a class.

Doing this implementation (see Exercise 7.15), we realize that the constructors are identical because their task in the present case is to store  $f$  and  $h$ . Object-orientation is now a natural next step: We can avoid duplicating the constructors by letting all the classes inherit the common constructor code. To this end, we introduce a superclass `Diff` and implement the different numerical differentiation rules in subclasses of `Diff`. Since the subclasses inherit their constructor, all they have to do is to provide a `__call__` method that implements the relevant differentiation formula.

Let us show what the superclass `Diff` looks like and how three subclasses implement the formulas (9.1)–(9.3):

```
class Diff:
    def __init__(self, f, h=1E-9):
        self.f = f
        self.h = float(h)

class Forward1(Diff):
    def __call__(self, x):
        f, h = self.f, self.h
        return (f(x+h) - f(x))/h

class Backward1(Diff):
    def __call__(self, x):
        f, h = self.f, self.h
        return (f(x) - f(x-h))/h

class Central2(Diff):
    def __call__(self, x):
        f, h = self.f, self.h
        return (f(x+h) - f(x-h))/(2*h)
```

These small classes demonstrate an important feature of object-orientation: code common to many different classes is placed in a superclass, and the subclasses add just the code that differs among the classes.

We can easily implement the formulas (9.4)–(9.6) by following the same method:

```
class Central4(Diff):
    def __call__(self, x):
        f, h = self.f, self.h
        return (4./3)*(f(x+h) - f(x-h)) / (2*h) - \
            (1./3)*(f(x+2*h) - f(x-2*h)) / (4*h)

class Central6(Diff):
    def __call__(self, x):
        f, h = self.f, self.h
        return (3./2)*(f(x+h) - f(x-h)) / (2*h) - \
            (3./5)*(f(x+2*h) - f(x-2*h)) / (4*h) + \
            (1./10)*(f(x+3*h) - f(x-3*h)) / (6*h)

class Forward3(Diff):
    def __call__(self, x):
```

```
f, h = self.f, self.h
return (-(1./6)*f(x+2*h) + f(x+h) - 0.5*f(x) - \
        (1./3)*f(x-h))/h
```

Here is a short example of using one of these classes to numerically differentiate the sine function<sup>4</sup>:

```
>>> from Diff import *
>>> from math import sin
>>> mycos = Central4(sin)
>>> # compute sin'(pi):
>>> mycos(pi)
-1.000000082740371
```

Instead of a plain Python function we may use an object with a `__call__` method, here exemplified through the function  $f(t; a, b, c) = at^2 + bt + c$ :

```
class Poly2:
    def __init__(self, a, b, c):
        self.a, self.b, self.c = a, b, c
    def __call__(self, t):
        return self.a*t**2 + self.b*t + self.c

f = Poly2(1, 0, 1)
dfdt = Central4(f)
t = 2
print "f'(%g)=%g" % (t, dfdt(t))
```

Let us examine the program flow. When Python encounters `dfdt = Central4(f)`, it looks for the constructor in class `Central4`, but there is no constructor in that class. Python then examines the superclasses of `Central4`, listed in `Central4.__bases__`. The superclass `Diff` contains a constructor, and this method is called. When Python meets the `dfdt(t)` call, it looks for `__call__` in class `Central4` and finds it, so there is no need to examine the superclass. This process of looking up methods of a class is called *dynamic binding*.

*Computer Science Remark.* Dynamic binding means that a name is bound to a function while the program is running. Normally, in computer languages, a function name is static in the sense that it is hard-coded as part of the function body and will not change during the execution of the program. This principle is known as static binding of function/method names. Object orientation offers the technical means to associate different functions with the same name, which yields a kind of magic for increased flexibility in programs. The particular function that the name refers to can be set at run-time, i.e., when the program is running, and therefore known as dynamic binding.

In Python, dynamic binding is a natural feature since names (variables) can refer to functions and therefore be dynamically bound dur-

<sup>4</sup> We have placed all the classes in the file `Diff.py` such that these classes constitute a module. In an interactive session or a small program, we must import the differentiation classes from the `Diff` module.

ing execution, just as any ordinary variable. To illustrate this point, let `func1` and `func2` be two Python functions of one argument, and consider the code

```
if input == 'func1':
    f = func1
elif input == 'func2':
    f = func2
y = f(x)
```

Here, the name `f` is bound to one of the `func1` and `func2` function objects while the program is running. This is a result of two features: (i) dynamic typing (so the contents of `f` can change), and (ii) functions being ordinary objects. The bottom line is that dynamic binding comes natural in Python, while it appears more like convenient magic in languages like C++, Java, and C#.

### 9.2.2 A Flexible Main Program

As a demonstration of the power of Python programming, we shall now write a program that accepts a function on the command-line, together with information about the difference type (centered, backward, or forward), the order of the approximation, and a value of the independent variable. The output from the program is the derivative of the given function. An example of the usage of the program goes like this:

---

```


Terminal


differentiate.py 'exp(sin(x))' Central 2 3.1
-1.04155573055

```

---

Here, we asked the program to differentiate  $f(x) = e^{\sin x}$  at  $x = 3.1$  with a central scheme of order 2 (using the `Central2` class in the `Diff` hierarchy).

We can provide any expression with `x` as input and request any scheme from the `Diff` hierarchy, and the derivative will be (approximately) computed. One great thing with Python is that the code is very short:

```
import sys
from Diff import *
from math import *
from scitools.StringFunction import StringFunction

formula = sys.argv[1]
f = StringFunction(formula)
difftype = sys.argv[2]
difforder = sys.argv[3]
classname = difftype + difforder
df = eval(classname + '(f)')
x = float(sys.argv[4])
print df(x)
```

Read the code line by line, and convince yourself that you understand what is going on. You may need to review Chapters 3.1.2 and 3.1.4.

One disadvantage is that the code above is limited to `x` as the name of the independent variable. If we allow a 5th command-line argument with the name of the independent variable, we can pass this name on to the `StringFunction` constructor, and suddenly our program works with any name for the independent variable!

```
varname = sys.argv[5]
f = StringFunction(formula, independent_variables=varname)
```

Of course, the program crashes if we do not provide five command-line arguments, and the program does not work properly if we are not careful with ordering of the command-line arguments. There is some way to go before the program is really user friendly, but that is beyond the scope of this chapter.

There are two strengths of the `differentiate.py` program: i) interactive specification of the function and the differentiation method, and ii) identical syntax for calling any differentiation method. With one line we create the subclass instance based on input strings. Many other popular programming languages (C++, Java, C#) cannot perform the `eval` operation while the program is running. The result is that we need `if` tests to turn the input string information into creation of subclass instances. Such type of code would look like this in Python:

```
if classname == 'Forward1':
    df = Forward1(f)
elif classname == 'Backward1':
    df = Backward1(f)
...
```

and so forth. This piece of code is very common in object-oriented systems and often put in a function that is referred to as a *factory function*. Factory functions can be made very compact in Python thanks to `eval`.

### 9.2.3 Extensions

The great advantage of sharing code via inheritance becomes obvious when we want to extend the functionality of a class hierarchy. It is possible to do this by adding more code to the superclass only. Suppose we want to be able to assess the accuracy of the numerical approximation to the derivative by comparing with the exact derivative, if available. All we need to do is to allow an extra argument in the constructor and provide an additional superclass method that computes the error in the numerical derivative. We may add this code to class `Diff`, or we may add it in a subclass `Diff2` and let the other classes for various

numerical differentiation formulas inherit from class `Diff2`. We follow the latter approach:

```
class Diff2(Diff):
    def __init__(self, f, h=1E-9, dfdx_exact=None):
        Diff.__init__(self, f, h)
        self.exact = dfdx_exact

    def error(self, x):
        if self.exact is not None:
            df_numerical = self(x)
            df_exact = self.exact(x)
            return df_exact - df_numerical

class Forward1(Diff2):
    def __call__(self, x):
        f, h = self.f, self.h
        return (f(x+h) - f(x))/h
```

The other subclasses, `Backward1`, `Central2`, and so on, must also be derived from `Diff2` to equip all subclasses with new functionality for perfectly assessing the accuracy of the approximation. No other modifications are necessary in this example, since all the subclasses can inherit the superclass constructor and the `error` method. Figure 9.2 shows a UML diagram of the new `Diff` class hierarchy.

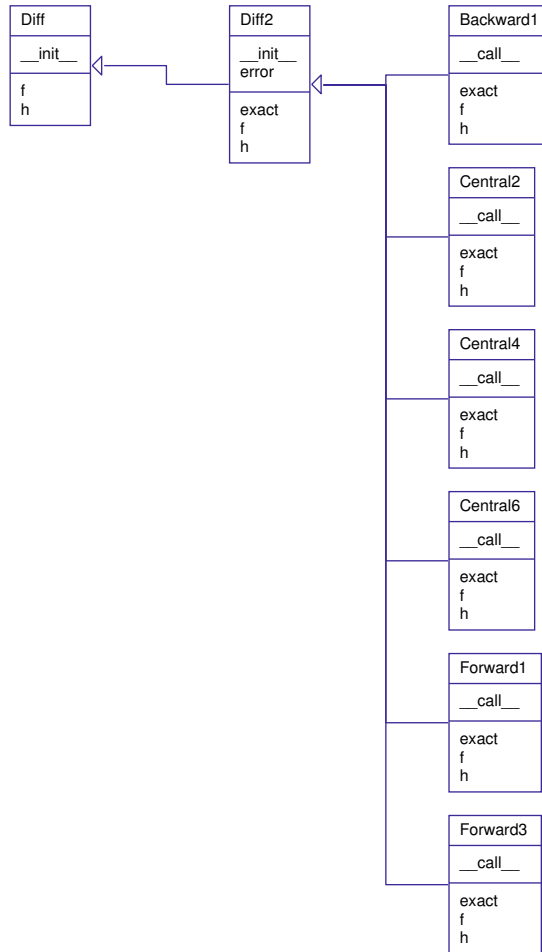
Here is an example of usage:

```
mycos = Forward1(sin, dfdx_exact=cos)
print 'Error in derivative is', mycos.error(x=pi)
```

The program flow of the `mycos.error(x=pi)` call can be interesting to follow. We first enter the `error` method in class `Diff2`, which then calls `self(x)`, i.e., the `__call__` method in class `Forward1`, which jumps out to the `self.f` function, i.e., the `sin` function in the `math` module in the present case. After returning to the `error` method, the next call is to `self.exact`, which is the `cos` function (from `math`) in our case.

*Application.* We can apply the methods in the `Diff2` hierarchy to get some insight into the accuracy of various difference formulas. Let us write out a table where the rows correspond to different  $h$  values, and the columns correspond to different approximation methods (except the first column which reflects the  $h$  value). The values in the table can be the numerically computed  $f'(x)$  or the error in this approximation if the exact derivative is known. The following function writes such a table:

```
def table(f, x, h_values, methods, dfdx=None):
    # print headline (h and class names for the methods):
    print '      h      ',
    for method in methods:
        print '%-15s' % method.__name__,
    print # newline
    for h in h_values:
        print '%10.2E' % h,
        for method in methods:
```



**Fig. 9.2** UML diagram of the Diff hierarchy for a series of differentiation formulas (Backward1, Central2, etc.).

```

if dfdx is not None:      # write error
    d = method(f, h, dfdx)
    output = d.error(x)
else:                     # write value
    d = method(f, h)
    output = d(x)
    print '%15.8E' % output,
    print # newline

```

The next lines tries three approximation methods on  $f(x) = e^{-10x}$  for  $x = 0$  and with  $h = 1, 1/2, 1/4, 1/16, \dots, 1/512$ :

```

from Diff2 import *
from math import exp

def f1(x):
    return exp(-10*x)

def df1dx(x):
    return -10*exp(-10*x)

```



```
table(f1, 0, [2**(-k) for k in range(10)],
      [Forward1, Central2, Central4], df1dx)
```

Note how convenient it is to make a list of class names – class names can be used as ordinary variables, and to print the class name as a string we just use the `__name__` attribute. The output of the main program above becomes

<code>h</code>	<code>Forward1</code>	<code>Central2</code>	<code>Central4</code>
1.00E+00	-9.00004540E+00	1.10032329E+04	-4.04157586E+07
5.00E-01	-8.01347589E+00	1.38406421E+02	-3.48320240E+03
2.50E-01	-6.32833999E+00	1.42008179E+01	-2.72010498E+01
1.25E-01	-4.29203837E+00	2.81535264E+00	-9.79802452E-01
6.25E-02	-2.56418286E+00	6.63876231E-01	-5.32825724E-02
3.12E-02	-1.41170013E+00	1.63556996E-01	-3.21608292E-03
1.56E-02	-7.42100948E-01	4.07398036E-02	-1.99260429E-04
7.81E-03	-3.80648092E-01	1.01756309E-02	-1.24266603E-05
3.91E-03	-1.92794011E-01	2.54332554E-03	-7.76243120E-07
1.95E-03	-9.70235594E-02	6.35795004E-04	-4.85085874E-08

From one row to the next,  $h$  is halved, and from about the 5th row and onwards, the `Forward1` errors are also halved, which is consistent with the error  $\mathcal{O}(h)$  of this method. Looking at the 2nd column, we see that the errors are reduced to  $1/4$  when going from one row to the next, at least after the 5th row. This is also according to the theory since the error is proportional to  $h^2$ . For the last row with a 4th-order scheme, the error is reduced by  $1/16$ , which again is what we expect when the error term is  $\mathcal{O}(h^4)$ . What is also interesting to observe, is the benefit of using a higher-order scheme like `Central4`: with, for example,  $h = 1/128$  the `Forward1` scheme gives an error of  $-0.7$ , `Central2` improves this to  $0.04$ , while `Central4` has an error of  $-0.0002$ . More accurate formulas definitely give better results<sup>5</sup>. The test example shown here is found in the file `Diff2_examples.py`.

#### 9.2.4 Alternative Implementation via Functions

Could we implement the functionality offered by the `Diff` hierarchy of objects by using plain functions and no object orientation? The answer is “yes, almost”. What we have to pay for a pure function-based solution is a less friendly user interface to the differentiation functionality: More arguments must be supplied in function calls, because each difference formula, now coded as a straight Python function, must get  $f(x)$ ,  $x$ , and  $h$  as arguments. In the class version we first store  $f$  and  $h$  as attributes in the constructor, and every time we want to compute the derivative, we just supply  $x$  as argument.

A Python function for implementing numerical differentiation reads

<sup>5</sup> Strictly speaking, it is the fraction of the work and the accuracy that counts: `Central4` needs four function evaluations, while `Central2` and `Forward1` only needs two.

```
def central2_func(f, x, h=1.0E-9):
    return (f(x+h) - f(x-h))/(2*h)
```

The usage demonstrates the difference from the class solution:

```
mycos = central2_func(sin, pi, 1E-6)
# compute sin'(pi):
print "g'(%g)=%g (exact value is %g)" % (pi, mycos, cos(pi))
```

Now, `mycos` is a number, not a callable object. The nice thing with the class solution is that `mycos` appeared to be a standard Python function whose mathematical values equal the derivative of the Python function `sin(x)`. But does it matter whether `mycos` is a function or a number? Yes, it matters if we want to apply the difference formula twice to compute the second-order derivative. When `mycos` is a callable object of type `Central2`, we just write

```
mysin = Central2(mycos)
# or
mysin = Central2(Central2(sin))

# compute g''(pi):
print "g''(%g)=%g" % (pi, mysin(pi))
```

With the `central2_func` function, this composition will not work. Moreover, when the derivative is an object, we can send this object to any algorithm that expects a mathematical function, and such algorithms include numerical integration, differentiation, interpolation, ordinary differential equation solvers, and finding zeros of equations, so the applications are many.

### 9.2.5 Alternative Implementation via Functional Programming

As a conclusion of the previous section, the great benefit of the object-oriented solution in Chapter 9.2.1 is that one can have some subclass instance `d` from the `Diff` (or `Diff2`) hierarchy and write `d(x)` to evaluate the derivative at a point `x`. The `d(x)` call behaves as if `d` were a standard Python function containing a manually coded expression for the derivative.

The `d(x)` interface to the derivative can also be obtained by other and perhaps more direct means than object-oriented programming. In programming languages where functions are ordinary objects that can be referred to by variables, as in Python, one can make a function that returns the right `d(x)` function according to the chosen numerical derivation rule. The code looks as this:

```
def differentiate(f, method, h=1.0E-9):
    h = float(h) # avoid integer division

    if method == 'Forward1':
```

```

def Forward1(x):
    return (f(x+h) - f(x))/h
    return Forward1

elif method == 'Backward1':
    def Backward1(x):
        return (f(x) - f(x-h))/h
        return Backward1
...

```

And the usage is like this:

```

mycos = differentiate(sin, 'Forward1')
mysin = differentiate(mycos, 'Forward1')
x = pi
print mycos(x), cos(x), mysin, -sin(x)

```

The surprising thing is that when we call `mycos(x)` we provide only `x`, while the function itself looks like

```

def Forward1(x):
    return (f(x+h) - f(x))/h
    return Forward1

```

How do the parameters `f` and `h` get their values when we call `mycos(x)`? There is some magic attached to the `Forward1` function, or literally, there are some variables attached to `Forward1`: this function “remembers” the values of `f` and `h` that existed as local variables in the `differentiate` function when the `Forward1` function was defined.

In computer science terms, the `Forward1` always has access to variables in the *scope* in which the function was defined. The `Forward1` function is what is known as a *closure* in some computer languages. Closures are much used in a programming style called *functional programming*. Two key features of functional programming is operations on lists (like list comprehensions) and returning functions from functions. Python supports functional programming, but we will not consider this programming style further in this book.

### 9.2.6 Alternative Implementation via a Single Class

Instead of making many classes or functions for the many different differentiation schemes, the basic information about the schemes can be stored in one table. With a single method in one single class can use the table information, and for a given scheme, compute the derivative. To do this, we need to reformulate the mathematical problem (actually by using ideas from Chapter 9.3.1).

A family of numerical differentiation schemes can be written

$$f'(x) \approx \sum_{i=-r}^r w_i f(x_i), \quad (9.7)$$

where  $w_i$  are weights and  $x_i$  are points. The  $2r+1$  points are symmetric around some point  $x$ :

$$x_i = x + ih, \quad i = -r, \dots, r.$$

The weights depend on the differentiation scheme. For example, the midpoint scheme (9.3) has

$$w_{-1} = -1, \quad w_0 = 0, \quad w_1 = 1.$$

Table 9.1 lists the values of  $w_i$  for different difference formulas. In this table we have set  $r = 4$ , which is sufficient for the schemes written up in this book.

Given a table of the  $w_i$  values, we can use (9.7) to compute the derivative. A faster, vectorized computation can have the  $x_i$ ,  $w_i$ , and  $f(x_i)$  values as stored in three vectors. Then  $\sum_i w_i f(x_i)$  can be interpreted as a dot product between the two vectors with components  $w_i$  and  $f(x_i)$ , respectively.

**Table 9.1** Weights in some difference schemes. The number after the nature of a scheme denotes the order of the schemes (for example, “central 2” is a central difference of 2nd order).

points	$x - 4h$	$x - 3h$	$x - 2h$	$x - h$	$x$	$x + h$	$x + 2h$	$x + 3h$	$x + 4h$
central 2	0	0	0	$-\frac{1}{2}$	0	$\frac{1}{2}$	0	0	0
central 4	0	0	$\frac{1}{12}$	$-\frac{2}{3}$	0	$\frac{2}{3}$	$-\frac{1}{12}$	0	0
central 6	0	$-\frac{1}{60}$	$\frac{3}{20}$	$-\frac{3}{4}$	0	$\frac{3}{4}$	$-\frac{3}{20}$	$\frac{1}{60}$	0
central 8	$\frac{1}{280}$	$-\frac{4}{105}$	$\frac{12}{60}$	$-\frac{4}{5}$	0	$\frac{4}{5}$	$-\frac{12}{60}$	$\frac{4}{105}$	$-\frac{1}{280}$
forward 1	0	0	0	0	1	1	0	0	0
forward 3	0	0	0	$-\frac{2}{6}$	$-\frac{1}{2}$	1	$-\frac{1}{6}$	0	0
backward 1	0	0	0	-1	1	0	0	0	0

A class with the table of weights as a static variable, a constructor, and a `__call__` method for evaluating the derivative via  $\sum_i w_i f(x_i)$  looks as follows:

```
class Diff3:
    table = {
        ('forward', 1):
            [0, 0, 0, 0, 1, 1, 0, 0, 0],
        ('central', 2):
            [0, 0, 0, -1./2, 0, 1./2, 0, 0, 0],
        ('central', 4):
            [0, 0, 1./12, -2./3, 0, 2./3, -1./12, 0, 0],
        ...
    }
    def __init__(self, f, h=1.0E-9, type='central', order=2):
        self.f, self.h, self.type, self.order = f, h, type, order
        self.weights = array(Diff2.table[(type, order)])

    def __call__(self, x):
        f_values = array([f(self.x+i*self.h) for i in range(-4,5)])
        return dot(self.weights, f_values)/self.h
```

Here we used `numpy`'s `dot(x, y)` function for computing the inner or dot product between two arrays `x` and `y`.

Class `Diff3` can be found in the file `Diff3.py`. Using class `Diff3` to differentiate the sine function goes like this:

```
import Diff3
mycos = Diff3.Diff3(sin, type='central', order=4)
print "sin'(pi):", mycos(pi)
```

*Remark.* The downside of class `Diff3`, compared with the other implementation techniques, is that the sum  $\sum_i w_i f(x_i)$  contains many multiplications by zero for lower-order schemes. These multiplications are known to yield zero in advance so we waste computer resources on trivial calculations. Once upon a time, programmers would have been extremely careful to avoid wasting multiplications this way, but today arithmetic operations are quite cheap, especially compared to fetching data from the computer's memory. Lots of other factors also influence the computational efficiency of a program, but this is beyond the scope of this book.

## 9.3 Class Hierarchy for Numerical Integration

There are many different numerical methods for integrating a mathematical function, just as there are many different methods for differentiating a function. It is thus obvious that the idea of object-oriented programming and class hierarchies can be applied to numerical integration formulas in the same manner as we did in Chapter 9.2.

### 9.3.1 Numerical Integration Methods

First, we list some different methods for integrating  $\int_a^b f(x)dx$  using  $n$  evaluation points. All the methods can be written as

$$\int_a^b f(x)dx \approx \sum_{i=0}^{n-1} w_i f(x_i), \quad (9.8)$$

where  $w_i$  are weights and  $x_i$  are evaluation points,  $i = 0, \dots, n-1$ . The Midpoint method has

$$x_i = a + \frac{h}{2} + ih, \quad w_i = h, \quad h = \frac{b-a}{n}, \quad i = 0, \dots, n-1. \quad (9.9)$$

The Trapezoidal method has the points

$$x_i = a + ih, \quad h = \frac{b-a}{n-1}, \quad i = 0, \dots, n-1, \quad (9.10)$$

and the weights

$$w_0 = w_{n-1} = \frac{h}{2}, \quad w_i = h, \quad i = 1, \dots, n-2. \quad (9.11)$$

Simpson's rule has the same evaluation points as the Trapezoidal rule, but

$$h = 2 \frac{b-a}{n-1}, \quad w_0 = w_{n-1} = \frac{h}{6}, \quad (9.12)$$

$$w_i = \frac{h}{3} \text{ for } i = 2, 4, \dots, n-3, \quad (9.13)$$

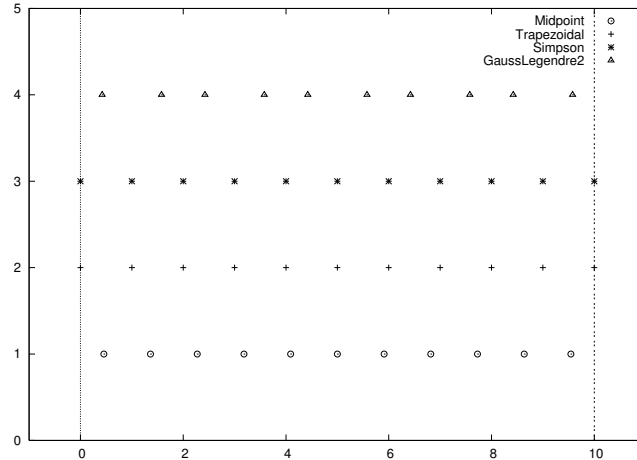
$$w_i = \frac{2h}{3} \text{ for } i = 1, 3, 5, \dots, n-2. \quad (9.14)$$

Note that  $n$  must be odd in Simpson's rule. A Two-Point Gauss-Legendre method takes the form

$$x_i = a + (i + \frac{1}{2})h - \frac{1}{\sqrt{3}} \frac{h}{2} \quad \text{for } i = 0, 2, 4, \dots, n-2, \quad (9.15)$$

$$x_i = a + (i + \frac{1}{2})h + \frac{1}{\sqrt{3}} \frac{h}{2} \quad \text{for } i = 1, 3, 5, \dots, n-1, \quad (9.16)$$

with  $h = 2(b-a)/n$ . Here  $n$  must be even. All the weights have the same value:  $w_i = h/2$ ,  $i = 0, \dots, n-1$ . Figure 9.3 illustrates how the points in various integration rules are distributed over a few intervals.



**Fig. 9.3** Illustration of the distribution of points for various numerical integration methods. The Gauss-Legendre method has 10 points, while the other methods have 11 points in  $[0, 10]$ .

### 9.3.2 Classes for Integration

We may store  $x_i$  and  $w_i$  in two NumPy arrays and compute the integral as  $\sum_{i=0}^{n-1} w_i f(x_i)$ . This operation can also be vectorized as a dot (inner) product between the  $w_i$  vector and the  $f(x_i)$  vector, provided  $f(x)$  is implemented in a vectorizable form.

We argued in Chapter 7.3.3 that it pays off to implement a numerical integration formula as a class. If we do so with the different methods from the previous section, a typical class looks like this:

```
class SomeIntegrationMethod:
    def __init__(self, a, b, n):
        # compute self.points and self.weights

    def integrate(self, f):
        s = 0
        for i in range(len(self.weights)):
            s += self.weights[i]*f(self.points[i])
        return s
```

Making such classes for many different integration methods soon reveals that all the classes contain common code, namely the `integrate` method for computing  $\sum_{i=0}^{n-1} w_i f(x_i)$ . Therefore, this common code can be placed in a superclass, and subclasses can just add the code that is specific to a certain numerical integration formula, namely the definition of the weights  $w_i$  and the points  $x_i$ .

Let us start with the superclass:

```
class Integrator:
    def __init__(self, a, b, n):
        self.a, self.b, self.n = a, b, n
        self.points, self.weights = self.construct_method()

    def construct_method(self):
        raise NotImplementedError('no rule in class %s' % \
                                  self.__class__.__name__)

    def integrate(self, f):
        s = 0
        for i in range(len(self.weights)):
            s += self.weights[i]*f(self.points[i])
        return s
```

As we have seen, we store the  $a$ ,  $b$ , and  $n$  data about the integration method in the constructor. Moreover, we compute arrays or lists `self.points` for the  $x_i$  points and `self.weights` for the  $w_i$  weights. All this code can now be inherited by all subclasses.

The initialization of points and weights is put in a separate method, `construct_method`, which is supposed to be implemented in each subclass, but the superclass provides a default implementation which tells the user that the method is not implemented. What happens is that when subclasses redefine a method, that method overrides the method inherited from the superclass. Hence, if we forget to redefine `construct_method` in a subclass, we will inherit the one from the superclass, and this method issues an error message. The construction of

this error message is quite clever in the sense that it will tell in which class the `construct_method` method is missing (`self` will be the subclass instance and its `__class__.__name__` is a string with the corresponding subclass name).

In computer science one usually speaks about *overloading* a method in a subclass, but the words redefining and overriding are also used. A method that is overloaded is said to be *polymorphic*. A related term, *polymorphism*, refers to coding with polymorphic methods. Very often, a superclass provides some default implementation of a method, and a subclass overloads the method with the purpose of tailoring the method to a particular application.

The `integrate` method is common for all integration rules, i.e., for all subclasses, so it can be inherited as it is. A vectorized version can also be added in the superclass to make it automatically available also in all subclasses:

```
def vectorized_integrate(self, f):
    return dot(self.weights, f(self.points))
```

Let us then implement a subclass. Only the `construct_method` method needs to be written. For the Midpoint rule, this is a matter of translating the formulas in (9.9) to Python:

```
class Midpoint(Integrator):
    def construct_method(self):
        a, b, n = self.a, self.b, self.n # quick forms
        h = (b-a)/float(n)
        x = linspace(a + 0.5*h, b - 0.5*h, n)
        w = zeros(len(x)) + h
        return x, w
```

Observe that we implemented directly a vectorized code. We could also have used (slow) loops and explicit indexing:

```
x = zeros(n)
w = zeros(n)
for i in range(n):
    x[i] = a + 0.5*h + i*h
    w[i] = h
```

Before we continue with other subclasses for other numerical integration formulas, we will have a look at the program flow when we use class `Midpoint`. Suppose we want to integrate  $\int_0^2 x^2 dx$  using 101 points:

```
def f(x): return x*x
m = Midpoint(0, 2, 101)
print m.integrate(f)
```

How is the program flow? The assignment to `m` invokes the constructor in class `Midpoint`. Since this class has no constructor, we invoke the inherited one from the superclass `Integrator`. Here attributes are



stored, and then the `construct_method` method is called. Since `self` is a `Midpoint` instance, it is the `construct_method` in the `Midpoint` class that is invoked, even if there is a method with the same name in the superclass. Class `Midpoint` overloads `construct_method` in the superclass. In a way, we “jump down” from the constructor in class `Integrator` to the `construct_method` in the `Midpoint` class. The next statement, `m.integrate(f)`, just calls the inherited `integral` method that is common to all subclasses.

A vectorized Trapezoidal rule can be implemented in another subclass with name `Trapezoidal`:

```
class Trapezoidal(Integrator):
    def construct_method(self):
        x = linspace(self.a, self.b, self.n)
        h = (self.b - self.a)/float(self.n - 1)
        w = zeros(len(x)) + h
        w[0] /= 2
        w[-1] /= 2
        return x, w
```

Observe how we divide the first and last weight by 2, using index 0 (the first) and -1 (the last) and the `/=` operator (`a /= b` is equivalent to `a = a/b`). Here also we could have implemented a scalar version with loops. The relevant code is in function `trapezoidal` in Chapter 7.3.3.

Class `Simpson` has a slightly more demanding rule, at least if we want to vectorize the expression, since the weights are of two types.

```
class Simpson(Integrator):
    def construct_method(self):
        if self.n % 2 != 1:
            print 'n=%d must be odd, 1 is added' % self.n
            self.n += 1
        x = linspace(self.a, self.b, self.n)
        h = (self.b - self.a)/float(self.n - 1)*2
        w = zeros(len(x))
        w[0:self.n:2] = h*1.0/3
        w[1:self.n-1:2] = h*2.0/3
        w[0] /= 2
        w[-1] /= 2
        return x, w
```

We first control that we have an odd number of points, by checking that the remainder of `self.n` divided by two is 1. If not, an exception could be raised, but for smooth operation of the class, we simply increase `n` so it becomes odd. Such automatic adjustments of input is not a rule to be followed in general. Wrong input is best notified explicitly. However, sometimes it is user friendly to make small adjustments of the input, as we do here, to achieve a smooth and successful operation. (In cases like this, a user might become uncertain whether the answer can be trusted if she (later) understands that the input should not yield a correct result. Therefore, do the adjusted computation, and provide a notification to the user about what has taken place.)

The computation of the weights `w` in class `Simpson` applies slices with stride (jump/step) 2 such that the operation is vectorized for speed. Recall that the upper limit of a slice is not included in the set, so `self.n-1` is the largest index in the first case, and `self.n-2` is the largest index in the second case. Instead of the vectorized operation of slices for computing `w`, we could use (slower) straight loops:

```
for i in range(0, self.n, 2):
    w[i] = h*1.0/3
for i in range(1, self.n-1, 2):
    w[i] = h*2.0/3
```

The points in the Two-Point Gauss-Legendre rule are slightly more complicated to calculate, so here we apply straight loops to make a safe first implementation:

```
class GaussLegendre2(Integrator):
    def construct_method(self):
        if self.n % 2 != 0:
            print 'n=%d must be even, 1 is subtracted' % self.n
            self.n -= 1
        nintervals = int(self.n/2.0)
        h = (self.b - self.a)/float(nintervals)
        x = zeros(self.n)
        sqrt3 = 1.0/sqrt(3)
        for i in range(nintervals):
            x[2*i] = self.a + (i+0.5)*h - 0.5*sqrt3*h
            x[2*i+1] = self.a + (i+0.5)*h + 0.5*sqrt3*h
        w = zeros(len(x)) + h/2.0
        return x, w
```

A vectorized calculation of `x` is possible by observing that the  $(i+0.5)h$  expression can be computed by `linspace`, and then we can add the remaining two terms:

```
m = linspace(0.5*h, (nintervals-1+0.5)*h, nintervals)
x[0:self.n-1:2] = m + self.a - 0.5*sqrt3*h
x[1:self.n:2] = m + self.a + 0.5*sqrt3*h
```

The array on the right-hand side has half the length of `x` ( $n/2$ ), but the length matches exactly the slice with stride 2 on the left-hand side.

### 9.3.3 Using the Class Hierarchy

To verify the implementation, we first try to integrate a linear function. All methods should compute the correct integral value regardless of the number of evaluation points:

```
def f(x):
    return x + 2

a = 2; b = 3; n = 4
for Method in Midpoint, Trapezoidal, Simpson, GaussLegendre2:
    m = Method(a, b, n)
    print m.__class__.__name__, m.integrate(f)
```

Observe how we simply list the class names as a tuple (comma-separated objects), and `Method` will in the `for` loop attain the values `Midpoint`, `Trapezoidal`, and so forth. For example, in the first pass of the loop, `Method(a, b, n)` is identical to `Midpoint(a, b, n)`.

The output of the test above becomes

```
Midpoint 4.5
Trapezoidal 4.5
n=4 must be odd, 1 is added
Simpson 4.5
GaussLegendre2 4.5
```

Since  $\int_2^3 (x+2)dx = \frac{9}{2} = 4.5$ , all methods passed this simple test.

A more challenging integral, from a numerical point of view, is

$$\int_0^1 \left(1 + \frac{1}{m}\right) t^{\frac{1}{m}} dt = 1.$$

To use any subclass in the `Integrator` hierarchy, the integrand must be a function of one variable only. For the present integrand, which depends on  $t$  and  $m$ , we use a class to represent it:

```
class F:
    def __init__(self, m):
        self.m = float(m)

    def __call__(self, t):
        m = self.m
        return (1 + 1/m)*t**(1/m)
```

We now ask the question: How much is the error in the integral reduced as we increase the number of integration points ( $n$ )? It appears that the error decreases exponentially with  $n$ , so if we want to plot the errors versus  $n$ , it is best to plot the logarithm of the error versus  $\ln n$ . We expect this graph to be a straight line, and the steeper the line is, the faster the error goes to zero as  $n$  increases. A common conception is to regard one numerical method as better than another if the error goes faster to zero as we increase the computational work (here  $n$ ).

For a given  $m$  and method, the following function computes two lists containing the logarithm of the  $n$  values, and the logarithm of the corresponding errors in a series of experiments:

```
def error_vs_n(f, exact, n_values, Method, a, b):
    log_n = [] # log of actual n values (Method may adjust n)
    log_e = [] # log of corresponding errors
    for n_value in n_values:
        method = Method(a, b, n_value)
        error = abs(exact - method.integrate(f))
        log_n.append(log(method.n))
        log_e.append(log(error))
    return log_n, log_e
```

We can plot the error versus  $n$  for several methods in the same plot and make one plot for each  $m$  value. The loop over  $m$  below makes such plots:

```

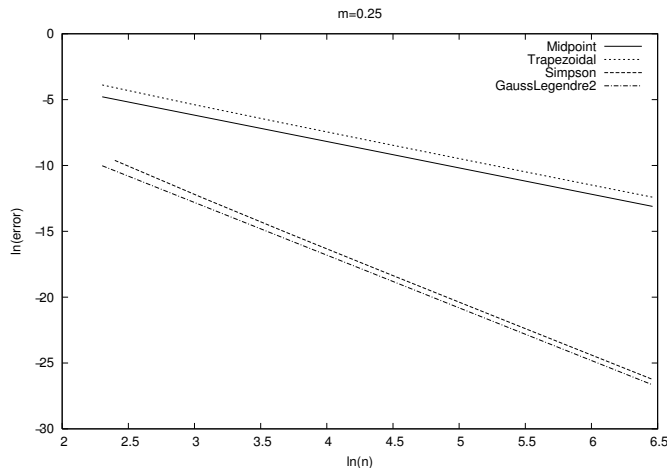
n_values = [10, 20, 40, 80, 160, 320, 640]
for m in 1./4, 1./8., 2, 4, 16:
    f = F(m)
    figure()
    for Method in Midpoint, Trapezoidal, \
        Simpson, GaussLegendre2:
        n, e = error_vs_n(f, 1, n_values, Method, 0, 1)
        plot(n, e); legend(Method.__name__); hold('on')
    title('m=%g' % m); xlabel('ln(n)'); ylabel('ln(error)')

```

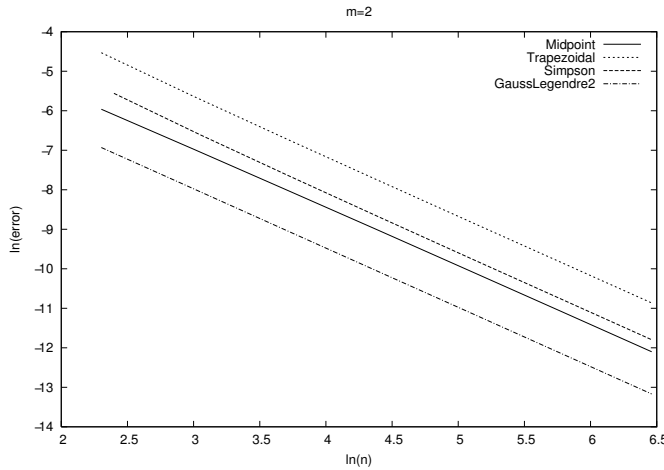
The code snippets above are collected in a function `test` in the `integrate.py` file.

The plots for  $m > 1$  look very similar. The plots for  $0 < m < 1$  are also similar, but different from the  $m > 1$  cases. Let us have a look at the results for  $m = 1/4$  and  $m = 2$ . The first,  $m = 1/4$ , corresponds to  $\int_0^1 5x^4 dx$ . Figure 9.4 shows that the error curves for the Trapezoidal and Midpoint methods converge more slowly compared to the error curves for Simpson's rule and the Gauss-Legendre method. This is the usual situation for these methods, and mathematical analysis of the methods can confirm the results in Figure 9.4.

However, when we consider the integral  $\int_0^1 \frac{3}{2}\sqrt{x}dx$ , ( $m = 2$ ) and  $m > 1$  in general, all the methods converge with the same speed, as shown in Figure 9.5. Our integral is difficult to compute numerically when  $m > 1$ , and the theoretically better methods (Simpson's rule and the Gauss-Legendre method) do not converge faster than the simpler methods. The difficulty is due to the infinite slope (derivative) of the integrand at  $x = 0$ .



**Fig. 9.4** The logarithm of the error versus the logarithm of integration points for integral  $5x^4$  computed by the Trapezoidal and Midpoint methods (upper two lines), and Simpson's rule and the Gauss-Legendre methods (lower two lines).



**Fig. 9.5** The logarithm of the error versus the logarithm of integration points for integral  $\int \sqrt[3]{x}$  computed by the Trapezoidal method and Simpson's rule (upper two lines), and Midpoint and Gauss-Legendre methods (lower two lines).

### 9.3.4 About Object-Oriented Programming

From an implementational point of view, the advantage of class hierarchies in Python is that we can save coding by inheriting functionality from a superclass. In programming languages where each variable must be specified with a fixed type, class hierarchies are particularly useful because a function argument with a special type also works with all subclasses of that type. Suppose we have a function where we need to integrate:

```
def do_math(arg1, arg2, integrator):
    ...
    I = integrator.integrate(myfunc)
    ...
```

That is, `integrator` must be an instance of some class, or a module, such that the syntax `integrator.integrate(myfunc)` corresponds to a function call, but nothing more (like having a particular type) is demanded.

This Python code will run as long as `integrator` has a method `integrate` taking one argument. In other languages, the function arguments are specified with a type, say in Java we would write

```
void do_math(double arg1, int arg2, Simpson integrator)
```

A compiler will examine all calls to `do_math` and control that the arguments are of the right type. Instead of specifying the integration method to be of type `Simpson`, one can in Java and other object-oriented languages specify `integrator` to be of the superclass type `Integrator`:

```
void do_math(double arg1, int arg2, Integrator integrator)
```

Now it is allowed to pass an object of any subclass type of `Integrator` as the third argument. That is, this method works with `integrator` of type `Midpoint`, `Trapezoidal`, `Simpson`, etc., not just one of them. Class hierarchies and object-oriented programming are therefore important means for parameterizing away types in languages like Java, C++, and C#. We do not need to parameterize types in Python, since arguments are not declared with a fixed type. Object-oriented programming is hence not so technically important in Python as in other languages for providing increased flexibility in programs.

Is there then any use for object-oriented programming beyond inheritance? The answer is yes! For many code developers object-oriented programming is not just a technical way of sharing code, but it is more a way of modeling the world, and understanding the problem that the program is supposed to solve. In mathematical applications we already have objects, defined by the mathematics, and standard programming concepts such as functions, arrays, lists, and loops are often sufficient for solving simpler problems. In the non-mathematical world the concept of objects is very useful because it helps to structure the problem to be solved. As an example, think of the phone book and message list software in a mobile phone. Class `Person` can be introduced to hold the data about one person in the phone book, while class `Message` can hold data related to an SMS message. Clearly, we need to know who sent a message so a `Message` object will have an associated `Person` object, or just a phone number if the number is not registered in the phone book. Classes help to structure both the problem and the program. The impact of classes and object-oriented programming on modern software development can hardly be exaggerated.

## 9.4 Class Hierarchy for Numerical Methods for ODEs

The next application targets numerical solution of ordinary differential equations. There is a jungle of such solution methods, a fact that suggests collecting the methods in a class hierarchy, just as we did for numerical differentiation and integration formulas.

### 9.4.1 Mathematical Problem

We may distinguish between two types of ordinary differential equations (ODEs): *scalar ODEs* and *systems of ODEs*. The former type involves one single equation,

$$\frac{du}{dt} = f(u, t), \quad u(0) = u_0, \quad (9.17)$$

with one unknown function  $u(t)$ . The latter type involves  $n$  equations with  $n$  unknown functions  $u^{(i)}(t)$ ,  $i = 0, \dots, n-1$ :

$$\frac{du^{(i)}}{dt} = f(u^{(0)}, u^{(1)}, \dots, u^{(n-1)}, t), \quad (9.18)$$

In addition, we need  $n$  initial conditions for a system with  $n$  equations and unknowns:

$$u^{(i)}(0) = u_0^{(i)}, \quad i = 0, \dots, n-1. \quad (9.19)$$

It is common to collect the functions  $u^{(0)}, u^{(1)}, \dots, u^{(n-1)}$  in a vector

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

and the initial conditions also in a vector

$$u_0 = (u_0^{(0)}, u_0^{(1)}, \dots, u_0^{(n-1)}).$$

In that case, (9.18) and (9.19) can be written as (9.17). We may take important advantage of this fact in an implementation: If the vectors are represented by Numerical Python arrays, the code we write for a scalar will very often work for arrays too. Unless you are quite familiar with systems of ODEs and array arithmetics, it can be a good idea to just think about scalar ODEs and that  $u(t)$  is a function of one variable when you read on. Later, you can come back and reread the text with systems of ODEs and  $u(t)$  as a vector (array) in mind. The text that follows and the program code are in fact independent of whether we solve scalar ODEs or systems of ODEs. This is quite a remarkable achievement, obtained by using a clever mathematical notation, where we do not distinguish between  $u$  as a scalar function or as a vector of functions, and the fact that scalar and array computations in Python look the same<sup>6</sup>.

*Example of a System of ODEs.* An oscillating spring-mass system can be governed by a second-order ODE (see (C.8) in Appendix C for derivation):

$$mu'' + \beta u' + ku = F(t), \quad u(0) = u_0, \quad u'(0) = 0.$$

The parameters  $m$ ,  $\beta$ , and  $k$  are known and  $F(t)$  is a prescribed function. This second-order equation can be rewritten as two first-order equations by introducing two functions (see Chapter B.5),

<sup>6</sup> The invisible difference between scalar ODEs and systems of ODEs is not only important for addressing *both* newcomers to ODEs and more experienced readers. The principle is very important for software development too: We can write code with scalar ODEs in mind and test this code. Afterwards, the code should also work immediately for systems and  $u(t)$  as a vector of functions!

$$u^{(0)}(t) = u(t), \quad u^{(1)}(t) = u'(t).$$

The unknowns are now the position  $u^{(0)}(t)$  and the velocity  $u^{(1)}(t)$ . We can then create equations where the derivative of the two new primary unknowns  $u^{(0)}$  and  $u^{(1)}$  appear alone on the left-hand side:

$$\frac{d}{dt}u^{(0)}(t) = u^{(1)}(t), \quad (9.20)$$

$$\frac{d}{dt}u^{(1)}(t) = m^{-1}(F(t) - \beta u^{(1)} - ku^{(0)}). \quad (9.21)$$

It is common to express such a system as  $u'(t) = f(u, t)$  where now  $u$  and  $f$  are vectors, here of length two:

$$\begin{aligned} u(t) &= (u^{(0)}(t), u^{(1)}(t)) \\ f(t, u) &= (u^{(1)}, m^{-1}(F(t) - \beta u^{(1)} - ku^{(0)})). \end{aligned} \quad (9.22)$$

#### 9.4.2 Numerical Methods

Numerical methods for ODEs compute approximations  $u_k$  to  $u$  at discrete time levels  $t_k$ ,  $k = 1, 2, 3, \dots$ . With a constant time step size  $\Delta t$  in time, we have  $t_k = k\Delta t$ . Some of the simplest, but also most widely used methods for ODEs are listed below.

1. The Forward Euler method:

$$u_{k+1} = u_k + \Delta t f(u_k, t_k). \quad (9.23)$$

2. The Midpoint method:

$$u_{k+1} = u_{k-1} + 2\Delta t f(u_k, t_k), \quad (9.24)$$

for  $k = 1, 2, \dots$ . For the first step, to compute  $u_1$ , the formula (9.24) involves  $u_{-1}$ , which is unknown, so here we must use another method, for instance, (9.23).

3. The 2nd-order Runge-Kutta method:

$$u_{k+1} = u_k + K_2 \quad (9.25)$$

where

$$K_1 = \Delta t f(u_k, t_k), \quad (9.26)$$

$$K_2 = \Delta t f(u_k + \frac{1}{2}K_1, t_k + \frac{1}{2}\Delta t). \quad (9.27)$$

4. The 4th-order Runge-Kutta method:

$$u_{k+1} = u_k + \frac{1}{6}(K_1 + 2K_2 + 2K_3 + K_4), \quad (9.28)$$



where

$$K_1 = \Delta t f(u_k, t_k), \quad (9.29)$$

$$K_2 = \Delta t f(u_k + \frac{1}{2}K_1, t_k + \frac{1}{2}\Delta t), \quad (9.30)$$

$$K_3 = \Delta t f(u_k + \frac{1}{2}K_2, t_k + \frac{1}{2}\Delta t), \quad (9.31)$$

$$K_4 = \Delta t f(u_k + K_3, t_k + \Delta t). \quad (9.32)$$

5. The Backward Euler method:

$$u_{k+1} = u_k + \Delta t f(u_{k+1}, t_{k+1}). \quad (9.33)$$

If  $f(u, t)$  is nonlinear in  $u$ , (9.33) constitutes a nonlinear equation in  $u_{k+1}$ , which must be solved by some method for nonlinear equations, say Newton's method (see Chapter 9.4.4 for more details).

The methods above are valid both for scalar ODEs and for systems of ODEs. In the system case, the quantities  $u$ ,  $u_k$ ,  $u_{k+1}$ ,  $f$ ,  $K_1$ ,  $K_2$ , etc., are vectors.

### 9.4.3 The ODE Solver Class Hierarchy

Chapter 7.4.2 presents a class `ForwardEuler` for implementing the Forward Euler scheme (9.23). Most of the code in this class is independent of the numerical method we use. In fact, we only need to change the `advance` method<sup>7</sup>. if we want another method, such as the 4-th order Runge-Kutta method, instead of the Forward Euler scheme. Copying the `ForwardEuler` class and editing just the `advance` method is considered bad programming practice, because we get two copies the general parts of class `ForwardEuler`. As we implement more schemes, we end up with a lot of copies of the same code. Correcting an error or improving the code in this general part then requires identical edits in several almost identical classes.

A good programming practice is to collect all the common code in a superclass. Subclasses can implement the `advance` method and share all other code.

*The Superclass.* We introduce class `ODESolver` as the superclass of all numerical methods for solving ODEs. Class `ODESolver` should provide all functionality that is common to all numerical methods for ODEs:

1. hold the solution  $u(t)$  at discrete time points in a list `u`
2. hold the corresponding time values `t`

<sup>7</sup> For more advanced methods than (9.23)–(9.33), especially so-called adaptive methods where  $\Delta t$  is automatically adjusted to gain an overall accuracy of the computations, there are more details that differ between various solution methods.

3. hold information about the  $f(u, t)$  function, i.e., a callable Python object `f(u, t)`
4. hold the (constant) time step  $\Delta t$  in an attribute `dt`
5. hold the time step number  $k$  in an attribute `k`
6. set the initial condition  $u_0$
7. implement the loop over all time steps

As already outlined, we implement the last point as two methods: `solve` for performing the time loop and `advance` for advancing the solution one time step. The latter method is empty in the superclass since the method is to be implemented by various subclasses for various numerical schemes.

A first version class `ODESolver` may follow the structure and contents of class `ForwardEuler` from Chapter 7.4.2:

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

    def advance(self):
        """Advance solution one time step."""
        raise NotImplementedError

    def set_initial_condition(self, u0, t0=0):
        self.u = []      # u[k] is solution at time t[k]
        self.t = []      # time levels in the solution process

        self.u.append(float(u0))
        self.t.append(float(t0))
        self.k = 0      # time level counter (k in formulas)

    def solve(self, T):
        """Advance solution in time until t <= T."""
        tnew = 0
        while tnew <= T:
            unew = self.advance()
            self.u.append(unew)
            tnew = self.t[-1] + self.dt
            self.t.append(tnew)
            self.k += 1
        return numpy.array(self.u), numpy.array(self.t)
```

Provided that `self.advance()` returns the solution at a new time level, this superclass contains everything needed to solve an ODE.

We can make an improvement of the `solve` method as explained in Exercise 7.26: The time loop is run as long as  $k \leq N$  and a user-defined function `terminate(u, t, k)` is `False`. By default, `terminate` can be the value `False`, and if desired, the programmer supplies some function that can be used to terminate the time loop on basis of the lists `u` and `t` and the time step counter `k`. For example, if we want to solve an ODE until the solution becomes zero, we can supply the function

```
def terminate(u, t, k):
    eps = 1.0E-6          # small number
    if abs(u[-1]) < eps:  # close enough to zero?
        return True
```

```

else:
    return False

```

The `solve` method then looks like

```

def solve(self, T, terminate=None):
    """
    Advance solution from t = t0 to t <= T, steps of dt
    as long as terminate(u,t,k) is False.
    terminate(u,t,k) is a user-given function
    returning True or False. By default, a terminate
    function which always returns False is used.
    """
    if terminate is None:
        terminate = lambda u, t, k: False
    self.k = 0
    tnew = 0
    while tnew <= T and \
        not terminate(self.u, self.t, self.k):

        unew = self.advance()

        self.u.append(unew)
        tnew = self.t[-1] + self.dt
        self.t.append(tnew)
        self.k += 1
    return numpy.array(self.u), numpy.array(self.t)

```

We use a default value of `None` to indicate that the user has not provided a `terminate` function. In that case, we make a `terminate` function that always returns `False` (see Chapter 2.2.11 for an explanation of using `lambda` for quickly defining a function).

*The Forward Euler Method.* Subclasses implement specific numerical formulas for numerical solution of ODEs in the `advance` method. For the Forward Euler the formula is given by (9.23). All data we need for this formula are stored as attributes by the superclass. First we load these data into variables with shorter names, to avoid the lengthy `self` prefix and obtain a notation closer to the mathematics. Then we apply the formula (9.23), and finally we return the new value:

```

class ForwardEuler(ODESolver):
    def advance(self):
        u, dt, f, k, t = \
            self.u, self.dt, self.f, self.k, self.t[-1]

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

```

A remark is worth mentioning: When we extract attributes to local variables with short names, we can only use these local variables for reading values, not setting values. For example, if we do a `k += 1` to update the time step counter, that increased value is not reflected in `self.k` (which is the “official” counter). Extracting class attributes in local variables is done for getting the code closer to the mathematics, but has a danger of introducing bugs that might be hard to track down.

*Systems of ODEs.* The codes for solving ODEs have up to now been written for scalar ODEs, not systems. However, we can with very small adjustments make all the code work with systems as well. In an ODE system,  $f(u[k], t)$  returns a list or array, depending on what the user prefers when implementing the right-hand side function. If a list is returned, we face a problem with  $dt*f(u[k], t)$  since multiplication of a float and a list is not defined. Therefore, we should automatically convert all right-hand sides to arrays. This is tedious to do inside the numerical algorithm. It is better to do it once and for all by redefining `self.f` in the constructor: we let `self.f` be a function that calls the user-given `f` and then feeds the returned list or array to `numpy.asarray` to ensure that we have an array to compute with. The `asarray` function does nothing if the argument is already an array. The following adjustment is then needed in the constructor:

```
def f_wrapper(u, t):
    return numpy.asarray(f(u, t), float)
self.f = f_wrapper

# or just
self.f = lambda u, t: numpy.asarray(f(u, t), float)
```

No other modifications are necessary for the ODE solvers to work perfectly with systems of ODEs, although we had only scalar ODEs in mind when we wrote the code!

*The 4th-order Runge-Kutta Method.* Below is an implementation of the 4th-order Runge-Kutta method (9.28):

```
class RungeKutta4(ODESolver):
    def advance(self):
        u, dt, f, k, t = \
            self.u, self.dt, self.f, self.k, self.t[-1]
        dt2 = dt/2.0
        K1 = dt*f(u[k], t)
        K2 = dt*f(u[k] + 0.5*K1, t + dt2)
        K3 = dt*f(u[k] + 0.5*K2, t + dt2)
        K4 = dt*f(u[k] + K3, t + dt)
        unew = u[k] + (1/6.0)*(K1 + 2*K2 + 2*K3 + K4)
        return unew
```

As long as the right-hand side function `f` is guaranteed to return an array in the case we want to solve an ODE system, the implemented Runge-Kutta method works for systems of ODEs as well as for scalar ODEs.

It is left as an exercise to implement other numerical methods in the `ODESolver` class hierarchy (see Exercises 9.27 and 9.28). However, the Backward Euler method (9.33) requires a much more advanced implementation than the other methods so that particular method deserves its own section.

### 9.4.4 The Backward Euler Method

The Backward Euler scheme (9.33) leads in general to a *nonlinear* equation at a new time level, while all the other schemes listed in Chapter 9.4.2 has a simple formula for a new  $u_{k+1}$  value. We see that (9.33) gives an equation to be solved for  $u_{k+1}$  by rearranging

$$u_{k+1} = u_k + \Delta t f(u_{k+1}, t_{k+1})$$

to

$$F(u_{k+1}) \equiv u_{k+1} - \Delta t f(u_{k+1}, t_{k+1}) - u_k = 0.$$

We must solve the equation  $F(u_{k+1}) = 0$  with respect to  $u_{k+1}$ . It may be easier to see this, and later easier to implement method, if we introduce a new variable  $w$  for  $u_{k+1}$ . The equation to be solved is then

$$F(w) \equiv w - \Delta t f(w, t_k) - u_k = 0. \quad (9.34)$$

If now  $f(u, t)$  is a nonlinear function of  $u$ ,  $F(w)$  will also be a nonlinear function of  $w$ .

To solve  $F(w) = 0$  we can use the Bisection method from Chapter 3.6.2, Newton's method from Chapter 5.1.9, or the Secant method from Exercise 5.14. Here we apply Newton's method and the implementation given in `src/diffeq/Newton.py`. A disadvantage with Newton's method is that we need the derivative of  $F$  with respect to  $w$ , which requires the derivative  $\partial f(w, t)/\partial w$ . A quick solution is to use a numerical derivative. Class `Derivative` from Chapter 7.3.2.

We make a subclass `BackwardEuler`. As we need to solve  $F(w) = 0$  at every time step, we also need to implement the  $F(w)$  function. We can do this in a method, as in

```
class BackwardEuler:
    def F(self, w):
        return w - \
            self.dt*self.f(w, self.t[-1]) - self.u[self.k]
```

Alternatively, we can make `F` as a local function inside the `advance` method<sup>8</sup>:

```
def advance(self):
    u, dt, f, k, t = \
        self.u, self.dt, self.f, self.k, self.t[-1]

    def F(w):
        return w - dt*f(w, t) - u[k]

    dFdw = Derivative(F)
    w_start = u[k] + dt*f(u[k], t)
    unew, n, F_value = Newton(F, w_start, dFdw, N=30)
    if n >= 30:
```

<sup>8</sup> The local variables in the `advance` function, e.g., `dt` and `u`, act as “global” variables for the `F` function. Hence, when `F` is sent away to some `Newton` function, `F` remembers the values of `dt`, `f`, `t`, and `u`!

```

        print "Newton's failed to converge at t=%g "\
              "(%d iterations)" % (t, n)
    return unew

```

The  $F(w)$  now looks closer to the mathematics. There are also some other statements that deserve a comment. The derivative  $dF/dw$  is computed numerically by a class `Derivative`, which is a slight modification of the similar class in Chapter 7.3.2, because we now want to use a more accurate, centered formula:

```

class Derivative:
    def __init__(self, f, h=1E-9):
        self.f = f
        self.h = float(h)

    def __call__(self, x):
        f, h = self.f, self.h
        return (f(x+h) - f(x-h))/(2*h)

```

This code is included in the `ODESolver.py` file after class `BackwardEuler`.

The next step is to call Newton's method. For this purpose we need to import the `Newton` function from the `Newton` module. However, this module is not located in the same folder as the `ODESolver` module, since the latter is in `src/oo` while the former is in `src/diffeq`. To tell Python to look for modules in `src/diffeq`, we modify `sys.path` as explained in Chapter 3.5.3:

```

import sys, os
sys.path.insert(0, os.path.join(os.pardir, 'diffeq'))
from Newton import Newton

```

Note that `diffeq` is a subfolder of our parent folder so we specify `../diffeq` rather than the full and possibly complicated path to `diffeq`. The parent folder is available as `os.pardir` ("parent directory"), and `os.path.join` combines folders with the right delimiter (forward slash on Mac/Linux/Unix and backward slash on Windows).

Having the `Newton` function from Chapter 5.1.9 accessible in our `ODESolver.py`, we can make a call and supply our  $F$  function as the argument `f`, a start value for the iteration, here called `w_start`, as the argument `x`, and the derivative  $dF/dw$  for the argument `dfdx`. We rely on default values for the `epsilon` and `store` arguments, while the maximum number of iterations is set to `N=30`. The program is terminated if it happens that the number of iterations exceeds that value, because then the method has diverged, and we have not been able to compute the next  $u_{k+1}$  value.

The starting value for Newton's method must be chosen. As we expect the solution to not change much from one time level to the next,  $u_k$  could be a good initial guess. However, we can do better by using a simple Forward Euler step  $u_k + \Delta t f(u_k, t_k)$ , which is exactly what we do in the `advance` function above.

Since Newton's method always has the danger of converging slowly, it can be interesting to store the number of iterations at each time level as an attribute in the `BackwardEuler` class. We can easily insert extra statement for this purpose:

```
def advance(self):
    ...
    unew, n, F_value = Newton(F, w_start, dFdw, N=30)
    if k == 0:
        self.Newton_iter = []
    self.Newton_iter.append(n)
    ...
```

Note the need for creating an empty list (at the first call of `advance`) before we can append elements.

There is now one important question to ask: Will the `advance` method work for systems of ODEs? In that case,  $F(w)$  is a vector of functions. The implementation of `F` will work when `w` is a vector, because all the quantities involved in the formula are arrays or scalar variables. The `dFdw` instance will compute a numerical derivative of each component of the vector function `dFdw.f` (which is simply our `F` function). The call to the `Newton` function is more critical: It turns out that this function, as the algorithm behind it, works for scalar equations only. Newton's method can quite easily be extended to a system of nonlinear equations, but we do not consider that topic here. Instead we equip class `BackwardEuler` with a constructor that calls the `f` object and controls that the returned value is a float and not an array:

```
class BackwardEuler(ODESolver):
    def __init__(self, f, dt):
        ODESolver.__init__(self, f, dt)
        # make a sample call to check that f is a scalar function:
        value = f(1,1)
        if not isinstance(value, (int, float)):
            raise ValueError\
                ('f(u,t) must return float/int, not %s' % type(value))
```

Observe that we must explicitly call the superclass constructor and pass on the arguments `f` and `dt` to achieve the right storage and treatment of these arguments.

Understanding class `BackwardEuler` implies a good understanding of classes in general, a good understanding of numerical methods for ODEs, for numerical differentiation, and for finding roots of functions, and a good understanding on how to combine different code segments from different parts of the book. Therefore, if you have digested class `BackwardEuler`, you have all reasons to believe that you have digested the key topics of this book.

### 9.4.5 Verification

We use the same verification problem as in Chapter 7.4.3, namely a function  $u(t)$  that is linear in  $t$  and that will be exactly reproduced by any of our schemes. Choosing  $u(t) = 0.2t + 3$  with a corresponding  $f(u, t) = 0.2 + (u - 0.2t - 3)^5$ , we can write the following code for testing the Forward Euler, Runge-Kutta, and Backward Euler methods:

```
def _f1(u, t):
    return 0.2 + (u - _u_solution_f1(t))**5

def _u_solution_f1(t):
    """Exact u(t) corresponding to _f1 above."""
    return 0.2*t + 3

def _verify(f, exact):
    u0 = 3; dt = 0.4; T = 2.8
    for Method_class in ForwardEuler, RungeKutta4, BackwardEuler:
        method = Method_class(f, dt)
        method.set_initial_condition(u0)
        u, t = method.solve(T)
        print Method_class.__name__, ':\n', u
    u_exact = exact(t)
    print 'Exact:\n', u_exact
    print 'Backward Euler iterations:', method.Newton_iter

if __name__ == '__main__':
    _verify(_f1, _u_solution_f1)
```

The output shows that all numerical methods provide exact numbers:

```
ForwardEuler:
[ 3.    3.08  3.16  3.24  3.32  3.4   3.48  3.56]
RungeKutta4:
[ 3.    3.08  3.16  3.24  3.32  3.4   3.48  3.56]
BackwardEuler :
[ 3.    3.08  3.16  3.24  3.32  3.4   3.48  3.56]
Exact:
[ 3.    3.08  3.16  3.24  3.32  3.4   3.48  3.56]
```

This is a good indication that many parts of our code are correct. (For the Backward Euler method, the test is insufficient because the starting value, being the Forward Euler prediction, is exact. Therefore, Newton's method does not need any iterations! Changing the start value to  $u[k]$  results in a linear equation for  $w$  and a need for one Newton iteration.)

### 9.4.6 Application 1: $u' = u$

The perhaps simplest of all ODEs,  $u' = u$ , is our first target problem for the classes in the ODESolver hierarchy. The basic part of the application of class ForwardEuler goes as follows:

```
from ODESolver import *
from scitools.std import *

def f(u, t):
```



```

    return u

T = 3
dt = 0.1
method = ForwardEuler(f, dt)
method.set_initial_condition(1.0)
u, t = method.solve(N)
plot(t, u)

```

We can easily demonstrate how superior the 4-th order Runge-Kutta method is for this equation when the time step is bigger ( $\Delta t = 1$ ):

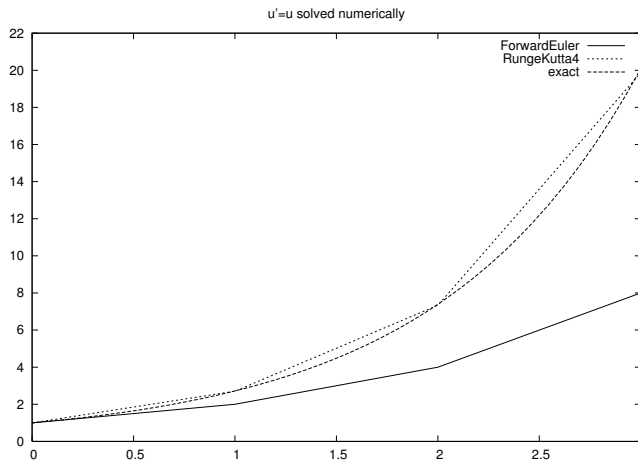
```

dt = 1
figure()
for Method_class in ForwardEuler, RungeKutta4:
    method = Method_class(f, dt)
    method.set_initial_condition(1)
    u, t = method.solve(T)
    plot(t, u)
    legend('%s' % method.__name__)
    hold('on')

t = linspace(0, T, 41) # finer resolution
plot(t, u_exact)
legend('exact')

```

Figure 9.6 shows the plot. The complete program can be found in the file `app1_exp.py`.



**Fig. 9.6** Comparison of the Forward Euler and the 4-th order Runge-Kutta method for solving  $u' = u$  for  $t \in [0, 3]$  and a long time step  $\Delta t = 1$ .

#### 9.4.7 Application 2: The Logistic Equation

The logistic ODE (B.23) is copied here for convenience:

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

The right-hand side contains the parameters  $\alpha$  and  $R$ . As emphasized in Chapters 7.1.1–7.1.2, the “right” way to code the right-hand side is then to make a class where  $\alpha$  and  $R$  are attributes, and where a `__call__` method evaluates the formula for the right-hand side. Such code is explained in Chapter 7.4.4.

However, by a mathematical simplification we can remove the  $\alpha$  and  $R$  parameters from the ODE and thereby simplify the ODE and also the implementation of the right-hand side. The simplification consists in scaling the independent and dependent variables, which is advantageous to do anyway if the goal is to understand more of the model equation and its solution. The scaling consists in introducing new variables

$$v = \frac{u}{R}, \quad \tau = \alpha t.$$

Inserting  $u = Rv$  and  $t = \tau/\alpha$  in the equation gives

$$\frac{dv}{d\tau} = v(1 - v), \quad v(0) = \frac{u_0}{R}.$$

Assume that we start with a small population, say  $u_0/R = 0.05$ . Amazingly, there are no more parameters in the equation for  $v(\tau)$ . That is, we can solve for  $v$  once and for all, and then recover  $u(t)$  by

$$u(t) = Rv(\alpha t).$$

Geometrically, the transformation from  $v$  to  $u$  is just a stretching of the two axis in the coordinate system.

We can compute  $v(\tau)$  by the 4-th order Runge-Kutta method in a program:

```
v0 = 0.05
dtau = 0.05
T = 10
method = RungeKutta4(lambda v, tau: v*(1-v), dtau)
method.set_initial_condition(v0)
v, tau = method.solve(T)
```

Observe that we use a lambda function (Chapter 2.2.11) to save some typing of a separate function for the right-hand side of the ODE. Now we need to run the program only once to compute  $v(t)$ , and from this solution we can easily create the solution  $u(t)$ , represented in terms of `u` and `t` arrays, by

```
t = alpha*tau
u = R*v
```

Below we make a plot to show how the  $u(t)$  curve varies with  $\alpha$ :

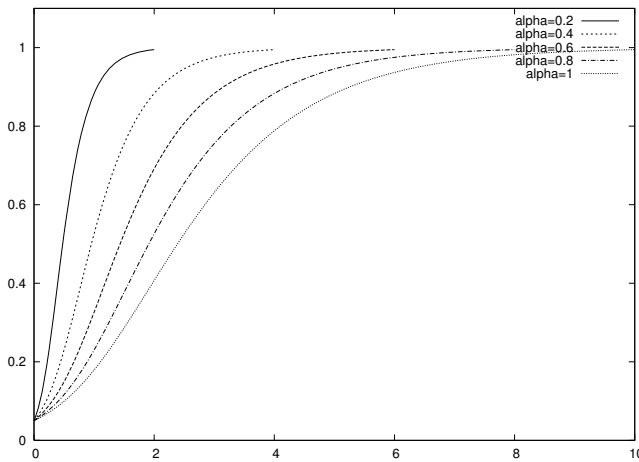
```
def ut(alpha, R):
    return alpha*tau, R*v

figure()
```

```
for alpha in linspace(0.2, 1, 5):
    t, u = ut(alpha, R=1)
    plot(t, u, legend='alpha=%g' % alpha)
    hold('on')
```

The resulting plot appears in Figure 9.7. Without the scaling, we would need to solve the ODE for each desired  $\alpha$  value. Furthermore, with the scaling we understand better that the influence of  $\alpha$  is only to stretch the  $t$  axis, or equivalently, stretch the curve along the  $t$  axis.

The complete program for this example is found in the file `app2_logistic.py`.



**Fig. 9.7** Solution of the logistic equation  $u' = \alpha u \left(1 - \frac{u}{R}\right)$  by the 4-th order Runge-Kutta method for various choices of  $\alpha$ .

### 9.4.8 Application 3: An Oscillating System

The motion of a box attached to a spring (Appendix C) can be modeled by two first-order differential equations as listed in (9.22) and repeated here for convenience:

$$\begin{aligned}\frac{du^{(0)}}{dt} &= u^{(1)}, \\ \frac{du^{(1)}}{dt} &= w''(t) + g - m^{-1}\beta u^{(1)} - m^{-1}ku^{(0)}.\end{aligned}$$

We now have a system of two ODEs, and the unknown is a vector containing the two functions, and the right-hand side  $f$  is also a vector with two components.

The code related to this example is found in `app3_osc.py`. Because our right-hand side  $f$  contains several parameters, we implement it as a class with the parameters as attributes and a `__call__` method for

returning the 2-vector  $f$ . We assume that the user of the class supplies the  $w(t)$  function, so it is natural to compute  $w''(t)$  by a finite difference formula.

```
class OscSystem:
    def __init__(self, m, beta, k, g, w):
        self.m, self.beta, self.k, self.g, self.w = \
            float(m), float(beta), float(k), float(g), w

    def __call__(self, u, t):
        u0, u1 = u
        m, beta, k, g, w = \
            self.m, self.beta, self.k, self.g, self.w
        # use a finite difference for w''(t):
        h = 1E-5
        ddw = (w(t+h) - 2*w(t) + w(t-h))/(2*h)
        f = [u1, ddw + g - beta/m*u1 - k/m*u0]
        return f
```

A simple test case arises if we set  $m = k = 1$  and  $\beta = g = w = 0$ :

$$\begin{aligned}\frac{du^{(0)}}{dt} &= u^{(1)}, \\ \frac{du^{(1)}}{dt} &= -u^{(0)}.\end{aligned}$$

Suppose that  $u^{(0)}(0) = 1$  and  $u^{(1)}(0) = 0$ . An exact solution is then

$$u^{(0)}(t) = \cos t, \quad u^{(1)}(t) = -\sin t.$$

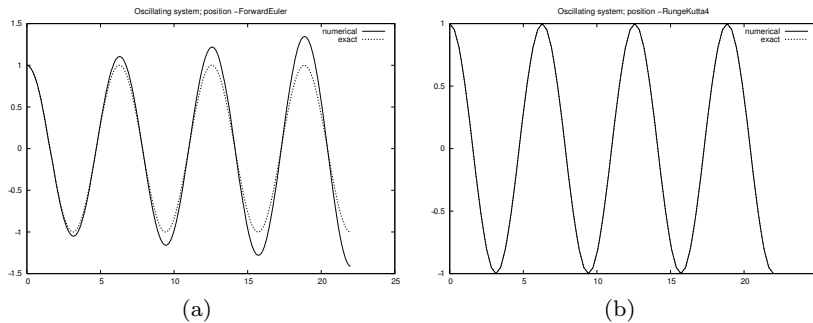
We can use this case to check how the Forward Euler method compares with the 4-th order Runge-Kutta method:

```
f = OscSystem(1.0, 0.0, 1.0, 0.0, lambda t: 0)
u_init = [1, 0] # initial condition
T = 7*pi
for Method_class in ForwardEuler, RungeKutta4:
    # let ForwardEuler dt be 1/10 of the RungeKutta dt:
    if Method_class == ForwardEuler:
        dt = 2*pi/200
    elif Method_class == RungeKutta4:
        dt = 2*pi/20
    method = Method_class(f, dt)
    method.set_initial_condition(u_init)
    u, t = method.solve(T)

    # u is an array of [u0,u1] pairs for each time level,
    # get the u0 values from u for plotting:
    u0_values = u[:, 0]
    u1_values = u[:, 1]
    u0_exact = cos(t)
    u1_exact = -sin(t)
    figure()
    alg = Method_class.__name__ # (class) name of algorithm
    plot(t, u0_values, 'r-',
         t, u0_exact, 'b-',
         legend=('numerical', 'exact'),
         title='Oscillating system; position - %s' % alg,
         hardcopy='tmp_oscsystem_pos_%s.eps' % alg)
    figure()
    plot(t, u1_values, 'r-',
```

```
t, u1_exact, 'b-',
legend=('numerical', 'exact'),
title='Oscillating system; velocity - %s' % alg,
hardcopy='tmp_oscsystem_vel_%s.eps' % alg)
```

For this particular application it turns out that the 4-th order Runge-Kutta is very accurate, even with few (20) time steps per oscillation (period). Unfortunately, the Forward Euler method leads to a solution with increasing amplitude in time. Figure 9.8 contains a comparison between the two methods. Note that the Forward Euler method uses 10 times as many time steps as the 4-th order Runge-Kutta method and is still much less accurate. A very much smaller time step is needed to limit the growth of the Forward Euler scheme for oscillating systems.



**Fig. 9.8** Solution of an oscillating system ( $u'' + u = 0$  formulated as system of two ODEs) by (a) the Forward Euler method with  $\Delta t = 2\pi/200$ ; and (b) the 4-th order Runge-Kutta method with  $\Delta t = 2\pi/20$ .

#### 9.4.9 Application 4: The Trajectory of a Ball

Exercise 1.14 derives the following two second-order differential equations for the motion of a ball (neglecting air resistance):

$$\frac{d^2x}{dt^2} = 0, \quad (9.35)$$

$$\frac{d^2y}{dt^2} = -g, \quad (9.36)$$

where  $(x, y)$  is the position of the ball ( $x$  is a horizontal measure and  $y$  is a vertical measure), and  $g$  is the acceleration of gravity. To use numerical methods for first-order equations, we must rewrite the system of two second-order equations as a system of four first-order equations. This is done by introducing to new unknowns, the velocities  $v_x = dx/dt$  and  $v_y = dy/dt$ . We then have the first-order system of ODEs

$$\frac{dx}{dt} = v_x, \quad (9.37)$$

$$\frac{dv_x}{dt} = 0, \quad (9.38)$$

$$\frac{dy}{dt} = v_y, \quad (9.39)$$

$$\frac{dv_y}{dt} = -g. \quad (9.40)$$

The initial conditions are

$$x(0) = 0, \quad (9.41)$$

$$v_x(0) = v_0 \cos \theta, \quad (9.42)$$

$$y(0) = y_0, \quad (9.43)$$

$$v_y(0) = v_0 \sin \theta, \quad (9.44)$$

where  $v_0$  is the initial magnitude of the velocity of the ball. The initial velocity has a direction that makes the angle  $\theta$  with the horizontal.

The code related to this example is found in `app4_ball.py`. A function returning the right-hand side of our ODE system reads

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

The main program for solving the ODEs can be set up as

```
v0 = 5
theta = 80*pi/180
u0 = [0, v0*cos(theta), 0, v0*sin(theta)]
T = 1.2
dt = 0.01
method = ForwardEuler(f, dt)
method.set_initial_condition(u0, 0)
u, t = method.solve(T)
```

Now, `u` is an array of 4-arrays `[x, vx, y, vy]`. Say we want to plot `x` as a function of time. We then have to extract all the `x` values as the first column in the two-dimensional `u` array:

```
x_values = u[:,0]
# or (slower):
x_values = array([x for x, vx, y, vy in u])
plot(t, x_values)
```

When a plot of the trajectory is desired, we need to plot the `y` coordinates of the ball against the `x` coordinates:

```
x_values = u[:,0]
y_values = u[:,2]
plot(x_values, y_values)
```

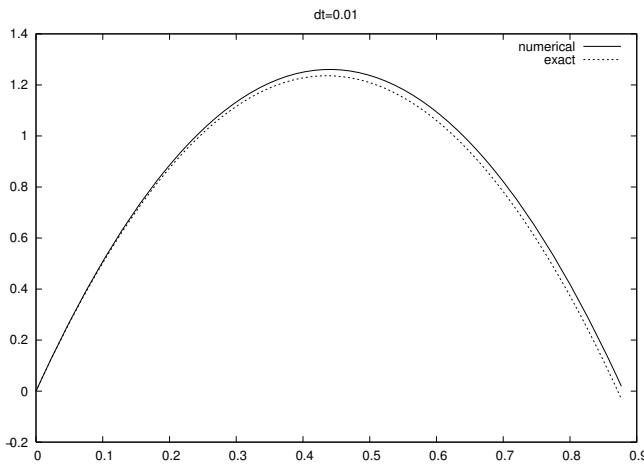
The exact solution is given by (1.5), so we can easily assess the accuracy of the numerical solution:

```
def exact(x):
    g = 9.81; y0 = u0[2]
    return x*tan(theta) - g*x**2/(2*v0**2)*1/(cos(theta))**2 + y0

plot(x_values, y_values, "r-",
     x_values, exact(x_values), "b-",
     legend=("numerical", "exact"),
     title="dt=%g" % dt)
```

Figure 9.9 shows a comparison of the numerical and the exact solution in this simple test problem. Note that even if we are just interested in  $y$  as a function of  $x$ , we first need to solve the complete ODE system for the arrays  $x$ ,  $vx$ ,  $y$ ,  $vy$  before we have  $x$  and  $y$  and can plot these.

The real strength of the numerical approach is the ease with which we can add air resistance and lift to the system of ODEs. Insight in physics is necessary to derive what the additional terms are, but implementing the terms is trivial in our test program above.



**Fig. 9.9** The trajectory of a ball solved as a system of four ODEs by the Forward Euler method.

## 9.5 Class Hierarchy for Geometric Shapes

Our next examples concern drawing geometric shapes. We know from Chapter 4 how to draw curves  $y = f(x)$ , but the point now is to construct some convenient software tools for drawing squares, circles, arcs, springs, wheels, and other shapes. With these tools we can create figures describing physical systems, for instance. Classes are very suitable for implementing the software because each shape is naturally associ-

ated with a class, and the various classes are related to each other through a natural hierarchy.

### 9.5.1 Using the Class Hierarchy

Before we dive into implementation details, let us first decide upon the interface we want to have for drawing various shapes. We start out by defining a rectangular area in which we will draw our figures. This is done by

```
from shapes import *
set_coordinate_system(xmin=0, xmax=10, ymin=0, ymax=10)
```

A line from (0,0) to (1,1) is defined by

```
l1 = Line(start=(0,0), stop=(1,1)) # define line
l1.draw()          # make plot data
display()          # display the plot data
```

A rectangle whose lower left corner is at (0,1), and where the width is 3 and the height is 5, is constructed by

```
r1 = Rectangle(lower_left_corner=(0,1), width=3, height=5)
r1.draw()
display()
```

A circle with center at (5,2) and unit radius, along with a wheel, is drawn by the code

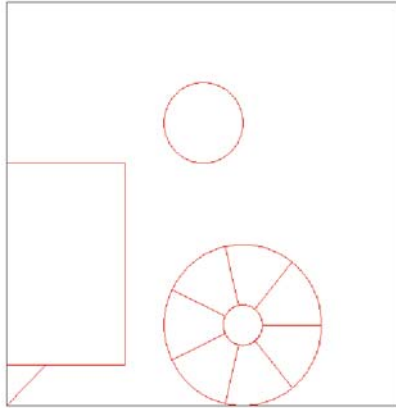
```
Circle(center=(5,7), radius=1).draw()
Wheel(center=(6,2), radius=2, inner_radius=0.5, nlines=7).draw()
display()
hardcopy('tmp') # create PNG file tmp.png
```

The latter line also makes a hardcopy of the figure in a PNG file. Figure 9.10 shows the resulting drawing after these commands.

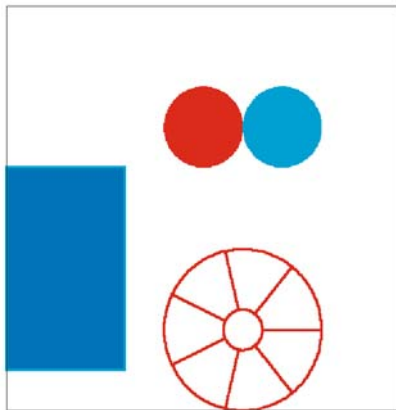
We can change the color and thickness of the lines and also fill circles, rectangles, etc. with a color. Figure 9.11 shows the result of the following example, where we first define elements in the figure and then adjust the line color and other properties prior to calling the `draw` methods:

```
r1 = Rectangle(lower_left_corner=(0,1), width=3, height=5)
c1 = Circle(center=(5,7), radius=1)
w1 = Wheel(center=(6,2), radius=2, inner_radius=0.5, nlines=7)
c2 = Circle(center=(7,7), radius=1)
filled_curves(True)
c1.draw()          # filled red circle
set_linecolor('blue')
r1.draw()          # filled blue rectangle
set_linecolor('aqua')
c2.draw()          # filled aqua/cyan circle
# add thick aqua line around rectangle:
filled_curves(False)
```





**Fig. 9.10** Result of a simple drawing session with shapes from the `Shape` class hierarchy.



**Fig. 9.11** Redrawing of some shapes from Figure 9.10 with some thicker lines and different colors.

```
set_linewidth(4)
r1.draw()
set_linecolor('red')
w1.draw()
display()
```

### 9.5.2 Overall Design of the Class Hierarchy

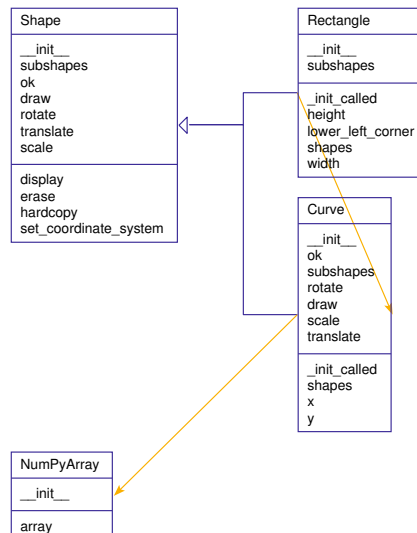
Let us have a class `Shape` as superclass for all specialized shapes. Class `Line` is a subclass of `Shape` and represents the simplest shape: a straight line between two points. Class `Rectangle` is another subclass of `Shape`, implementing the functionality needed to specify the four lines of a rectangle. Class `Circle` can be yet another subclass of `Shape`, or we may have a class `Arc` and let `Circle` be a subclass of `Arc` since a circle is an arc of 360 degrees. Class `Wheel` is also subclass of `Shape`, but it

contains naturally two `Circle` instances for the inner and outer circles, plus a set of `Line` instances going from the inner to the outer circles.

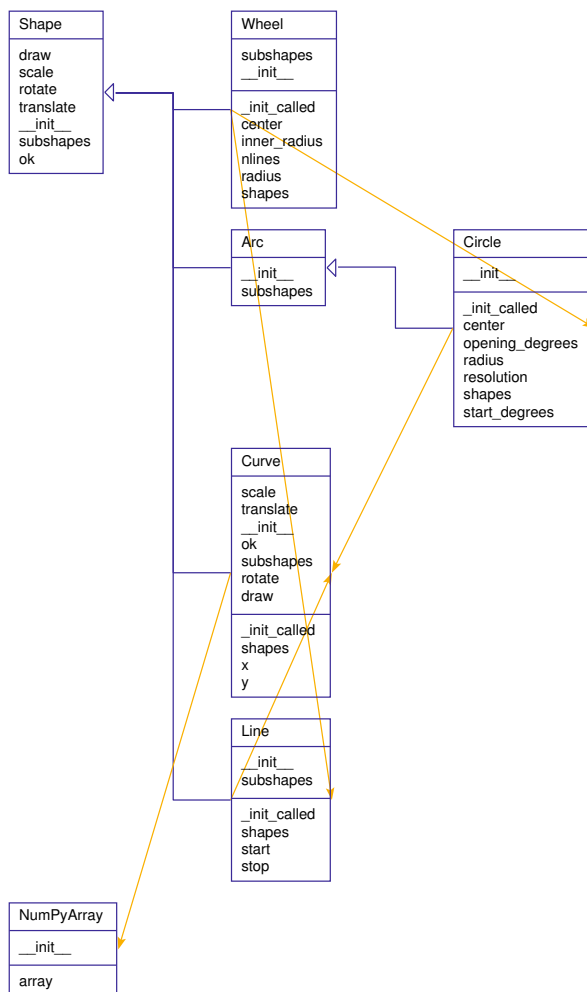
The discussion in the previous paragraph shows that a subclass in the `Shape` hierarchy typically contains a list of other subclass instances, *or* the shape is a primitive, such as a line, circle, or rectangle, where the geometry is defined through a set of  $(x, y)$  coordinates rather than through other `Shape` instances. It turns out that the implementation is simplest if we introduce a class `Curve` for holding a primitive shape defined by  $(x, y)$  coordinates. Then all other subclasses of `Shape` can have a list `shapes` holding the various instances of subclasses of `Shape` needed to build up the geometric object. The `shapes` attribute in class `Circle` will contain one `Curve` instance for holding the coordinates along the circle, while the `shapes` attribute in class `Wheel` contains two `Circle` instances and a number of `Line` instances. Figures 9.12 and 9.13 display two UML drawings of the `shapes` class hierarchy where we can get a view of how `Rectangle` and `Wheel` relate to other classes: the darkest arrows represent is-a relationship while the lighter arrows represent has-a relationship.

All instances in the `Shape` hierarchy must have a `draw` method. The `draw` method in class `Curve` plots the  $(x, y)$  coordinates as a curve, while the `draw` method in all other classes simply do a

```
for shape in self.shapes:
    shape.draw()
```



**Fig. 9.12** UML diagram of parts of the `shapes` hierarchy. Classes `Rectangle` and `Curve` are subclasses of `Shape`. The darkest arrow with the biggest arrowhead indicates inheritance and is-a relationship: `Rectangle` and `Curve` are both also `Shape`. The lighter arrow indicates has-a relationship: `Rectangle` has a `Curve`, and a `Curve` has a `NumPyArray`.



**Fig. 9.13** This is a variant of Figure 9.12 where we display how class `Wheel` relates to other classes in the `shapes` hierarchy. `Wheel` is a `Shape`, like `Arc`, `Line`, and `Curve`, but `Wheel` contains `Circle` and `Line` objects, while the `Circle` and `Line` objects have a `Curve`, which has a `NumPyArray`. We also see that `Circle` is a subclass of `Arc`.

### 9.5.3 The Drawing Tool

We have in Chapter 4 introduced the `Easyviz` tool for plotting graphs. This tool is quite well suited for drawing geometric shapes defined in terms of curves, but when drawing shapes we often want to skip tickmarks on the axis, labeling of the curves and axis, and perform other adjustments. Instead of using `Easyviz`, which aims at function plotting, we have decided to use a plotting tool directly and fine-tune the few commands we need for drawing shapes.

A simple plotting tool for shapes is based on `Gnuplot` and implemented in class `GnuplotDraw` in the file `GnuplotDraw.py`. This class has the following user interface:

```

class GnuplotDraw:
    def __init__(self, xmin, xmax, ymin, ymax):
        """Define the drawing area [xmin,xmax]x[ymin,ymax]."""

    def define_curve(self, x, y):
        """Define a curve with coordinates x and y (arrays)."""

    def erase(self):
        """Erase the current figure."""

    def display(self):
        """Display the figure."""

    def hardcopy(self, name):
        """Save figure in PNG file name.png."""

    def set_linecolor(self, color):
        """Change the color of lines."""

    def set_linewidth(self, width):
        """Change the line width (int, starts at 1)."""

    def filled_curves(self, on=True):
        """Fill area inside curves with current line color."""

```

One can easily make a similar class with an identical interface that applies another plotting package than Gnuplot to create the drawings. In particular, encapsulating the drawing actions in such a class makes it trivial to change the drawing program in the future. The program pieces that apply a drawing tool like GnuplotDraw remain the same. This is an important strategy to follow, especially when developing larger software systems.

#### 9.5.4 Implementation of Shape Classes

Our superclass `Shape` can naturally hold a coordinate system specification, i.e., the rectangle in which other shapes can be drawn. This area is fixed for all shapes, so the associated variables should be static and the method for setting them should also be static (see Chapter 7.7 for static attributes and methods). It is also natural that class `Shape` holds access to a drawing tool, in our case a `GnuplotDraw` instance. This object is also static. However, it can be an advantage to mirror the static attributes and methods as global variables and functions in the `shapes` modules. Users not familiar with static class items can drop the `Shape` prefix and just use plain module variables and functions. This is what we do in the application examples.

Class `Shape` defines an important method, `draw`, which just calls the `draw` method for all subshapes that build up the current shape.

Here is a brief view of class `Shape`<sup>9</sup>:

<sup>9</sup> We have for simplicity omitted the static attributes and methods. These can be viewed in the `shapes.py` file.

```
class Shape:
    def __init__(self):
        self.shapes = self.subshapes()
        if isinstance(self.shapes, Shape):
            self.shapes = [self.shapes] # turn to list

    def subshapes(self):
        """Define self.shapes as list of Shape instances."""
        raise NotImplementedError(self.__class__.__name__)

    def draw(self):
        for shape in self.shapes:
            shape.draw()
```

In class `Shape` we require the `shapes` attribute to be a list, but if the `subshape` method in subclasses returns just one instance, this is automatically wrapped in a list in the constructor.

First we implement the special case class `Curve`, which does not have `subshapes` but instead  $(x, y)$  coordinates for a curve:

```
class Curve(Shape):
    """General (x,y) curve with coordintes."""
    def __init__(self, x, y):
        self.x, self.y = x, y
        # turn to Numerical Python arrays:
        self.x = asarray(self.x, float)
        self.y = asarray(self.y, float)
        Shape.__init__(self)

    def subshapes(self):
        pass # geometry defined in constructor
```

The simplest ordinary `Shape` class is `Line`:

```
class Line(Shape):
    def __init__(self, start, stop):
        self.start, self.stop = start, stop
        Shape.__init__(self)

    def subshapes(self):
        x = [self.start[0], self.stop[0]]
        y = [self.start[1], self.stop[1]]
        return Curve(x,y)
```

The code in this class works with `start` and `stop` as tuples, lists, or arrays of length two, holding the end points of the line. The underlying `Curve` object needs only these two end points.

A rectangle is represented by a slightly more complicated class, having the lower left corner, the width, and the height of the rectangle as attributes:

```
class Rectangle(Shape):
    def __init__(self, lower_left_corner, width, height):
        self.lower_left_corner = lower_left_corner # 2-tuple
        self.width, self.height = width, height
        Shape.__init__(self)

    def subshapes(self):
        ll = self.lower_left_corner # short form
```

```

x = [ll[0], ll[0]+self.width,
     ll[0]+self.width, ll[0], ll[0]]
y = [ll[1], ll[1], ll[1]+self.height,
     ll[1]+self.height, ll[1]]
return Curve(x,y)

```

Class `Circle` needs many coordinates in its `Curve` object in order to display a smooth circle. We can provide the number of straight line segments along the circle as a parameter `resolution`. Using a default value of 180 means that each straight line segment approximates an arc of 2 degrees. This resolution should be sufficient for visual purposes. The set of coordinates along a circle with radius  $R$  and center  $(x_0, y_0)$  is defined by

$$x = x_0 + R \cos(t), \quad (9.45)$$

$$y = y_0 + R \sin(t), \quad (9.46)$$

for `resolution+1`  $t$  values between 0 and  $2\pi$ . The vectorized NumPy code for computing the coordinates becomes

```

t = linspace(0, 2*pi, self.resolution+1)
x = x0 + R*cos(t)
y = y0 + R*sin(t)

```

The complete `Circle` class is shown below:

```

class Circle(Shape):
    def __init__(self, center, radius, resolution=180):
        self.center, self.radius = center, radius
        self.resolution = resolution
        Shape.__init__(self)

    def subshapes(self):
        t = linspace(0, 2*pi, self.resolution+1)
        x0 = self.center[0]; y0 = self.center[1]
        R = self.radius
        x = x0 + R*cos(t)
        y = y0 + R*sin(t)
        return Curve(x,y)

```

We can also introduce class `Arc` for drawing the arc of a circle. Class `Arc` could be a subclass of `Circle`, extending the latter with two additional parameters: the opening of the arc (in degrees) and the starting  $t$  value in (9.45)–(9.46). The implementation of class `Arc` will then be almost a copy of the implementation of class `Circle`. The `subshapes` method will just define a different  $t$  array.

Another view is to let class `Arc` be a subclass of `Shape`, and `Circle` a subclass of `Arc`, since a circle is an arc of 360 degrees. Let us employ this idea:

```

class Arc(Shape):
    def __init__(self, center, radius,
                 start_degrees, opening_degrees, resolution=180):
        self.center = center
        self.radius = radius

```

```

        self.start_degrees = start_degrees*pi/180
        self.opening_degrees = opening_degrees*pi/180
        self.resolution = resolution
        Shape.__init__(self)

    def subshapes(self):
        t = linspace(self.start_degrees,
                     self.start_degrees + self.opening_degrees,
                     self.resolution+1)
        x0 = self.center[0]; y0 = self.center[1]
        R = self.radius
        x = x0 + R*cos(t)
        y = y0 + R*sin(t)
        return Curve(x,y)

class Circle(Arc):
    def __init__(self, center, radius, resolution=180):
        Arc.__init__(self, center, radius, 0, 360, resolution)

```

In this latter implementation, we save a lot of code in class `Circle` since all of class `Arc` can be reused.

Class `Wheel` may conceptually be a subclass of `Circle`. One circle, say the outer, is inherited and the subclass must have the inner circle as an attribute. Because of this “asymmetric” representation of the two circles in a wheel, we find it more natural to derive `Wheel` directly from `Shape`, and have the two circles as two attributes of type `Circle`:

```

class Wheel(Shape):
    def __init__(self, center, radius, inner_radius=None, nlines=10):
        self.center = center
        self.radius = radius
        if inner_radius is None:
            self.inner_radius = radius/5.0
        else:
            self.inner_radius = inner_radius
        self.nlines = nlines
        Shape.__init__(self)

```

If the radius of the inner circle is not defined (`None`) we take it as  $1/5$  of the radius of the outer circle. The wheel is naturally composed of two `Circle` instances and `nlines` `Line` instances:

```

    def subshapes(self):
        outer = Circle(self.center, self.radius)
        inner = Circle(self.center, self.inner_radius)
        lines = []
        t = linspace(0, 2*pi, self.nlines)
        Ri = self.inner_radius; Ro = self.radius
        x0 = self.center[0]; y0 = self.center[1]
        xinner = x0 + Ri*cos(t)
        yinner = y0 + Ri*sin(t)
        xouter = x0 + Ro*cos(t)
        youter = y0 + Ro*sin(t)
        lines = [Line((xi,yi),(xo,yo)) for xi, yi, xo, yo in \
                  zip(xinner, yinner, xouter, youter)]
        return [outer, inner] + lines

```

For the fun of it, we can implement other shapes, say a sine wave

$$y = m + A \sin kx, \quad k = 2\pi/\lambda,$$

where  $\lambda$  is the wavelength of the sine waves,  $A$  is the wave amplitude, and  $m$  is the mean value of the wave. The class looks like

```
class Wave(Shape):
    def __init__(self, xstart, xstop,
                  wavelength, amplitude, mean_level):
        self.xstart = xstart
        self.xstop = xstop
        self.wavelength = wavelength
        self.amplitude = amplitude
        self.mean_level = mean_level
        Shape.__init__(self)

    def subshapes(self):
        npoints = (self.xstop - self.xstart)/(self.wavelength/61.0)
        x = linspace(self.xstart, self.xstop, npoints)
        k = 2*pi/self.wavelength # frequency
        y = self.mean_level + self.amplitude*sin(k*x)
        return Curve(x,y)
```

With this and the previous example, you should be in a position to write your own subclasses. Exercises 9.35–9.39 suggest some smaller projects.

*Functions for Controlling Lines, Colors, etc.* The `shapes` module containing class `Shape` and all subclasses mentioned above, also offers some additional functions that do not depend on any particular shape:

- `display()` for displaying the defined figures so far (all figures whose `draw` method is called).
- `erase()` for erasing the current figure.
- `hardcopy(name)` for saving the current figure to a PNG file `name.png`.
- `set_linecolor(color)` for setting the color of lines, where `color` is a string like `'red'` (default), `'blue'`, `'green'`, `'aqua'`, `'purple'`, `'yellow'`, and `'black'`.
- `set_linewidth(width)` for setting the width of a line, measured as an integer (default is 2).
- `filled_curves(on)` for turning on (`on=True`) or off (`on=False`) whether the area inside a shape should be filled with the current line color.

Actually, the functions above are static methods in class `Shape` (cf. Chapter 7.7), and they are just mirrored as global functions<sup>10</sup> in the `shapes` module. Users without knowledge of static methods do not need to use the `Shape` prefix for reaching this functionality.

### 9.5.5 Scaling, Translating, and Rotating a Figure

The real power of object-oriented programming will be obvious in a minute when we, with a few lines of code, suddenly can equip *all* shape

<sup>10</sup> You can look into `shapes.py` to see how we automate the duplication of static methods as global functions.



objects with additional functionality for scaling, translating, and rotating the figure.

*Scaling.* Let us first treat the simplest of the three cases: scaling. For a `Curve` instance containing a set of  $n$  coordinates  $(x_i, y_i)$  that make up a curve, scaling by a factor  $a$  means that we multiply all the  $x$  and  $y$  coordinates by  $a$ :

$$x_i \leftarrow ax_i, \quad y_i \leftarrow ay_i, \quad i = 0, \dots, n-1.$$

Here we apply the arrow as an assignment operator. The corresponding Python implementation in class `Curve` reads

```
class Curve:
    ...
    def scale(self, factor):
        self.x = factor*self.x
        self.y = factor*self.y
```

Note here that `self.x` and `self.y` are Numerical Python arrays, so that multiplication by a scalar number `factor` is a vectorized operation.

In an instance of a subclass of `Shape`, the meaning of a method `scale` is to run through all objects in the list `self.shapes` and ask each object to scale itself. This is the same delegation of actions to subclass instances as we do in the `draw` method, and all objects, except `Curve` instances, can share the same implementation of the `scale` method. Therefore, we place the `scale` method in the superclass `Shape` such that all subclasses can inherit this method. Since `scale` and `draw` are so similar, we can easily implement the `scale` method in class `Shape` by copying and editing the `draw` method:

```
class Shape:
    ...
    def scale(self, factor):
        for shape in self.shapes:
            shape.scale(factor)
```

This is all we have to do in order to equip all subclasses of `Shape` with scaling functionality! But why is it so easy? All subclasses inherit `scale` from class `Shape`. Say we have a subclass instance `s` and that we call `s.scale(factor)`. This leads to calling the inherited `scale` method shown above, and in the `for` loop we call the `scale` method for each `shape` object in the `self.shapes` list. If `shape` is not a `Curve` object, this procedure repeats, until we hit a `shape` that is a `Curve`, and then the scaling on that set of coordinates is performed.

*Translation.* A set of coordinates  $(x_i, y_i)$  can be translated  $x$  units in the  $x$  direction and  $y$  units in the  $y$  direction using the formulas

$$x_i \leftarrow x + x_i, \quad y_i \leftarrow y + y_i, \quad i = 0, \dots, n-1.$$

The corresponding Python implementation in class `Curve` becomes

```
class Curve:
    ...
    def translate(self, x, y):
        self.x = x + self.x
        self.y = y + self.y
```

The translation operation for a shape object is very similar to the scaling and drawing operations. This means that we can implement a common method `translate` in the superclass `Shape`. The code is parallel to the `scale` method:

```
class Shape:
    ....
    def translate(self, x, y):
        for shape in self.shapes:
            shape.translate(x, y)
```

*Rotation.* Rotating a figure is more complicated than scaling and translating. A counter clockwise rotation of  $\theta$  degrees for a set of coordinates  $(x_i, y_i)$  is given by

$$\begin{aligned}\bar{x}_i &\leftarrow x_i \cos \theta - y_i \sin \theta, \\ \bar{y}_i &\leftarrow x_i \sin \theta + y_i \cos \theta.\end{aligned}$$

This rotation is performed around the origin. If we want the figure to be rotated with respect to a general point  $(x, y)$ , we need to extend the formulas above:

$$\begin{aligned}\bar{x}_i &\leftarrow x + (x_i - x) \cos \theta - (y_i - y) \sin \theta, \\ \bar{y}_i &\leftarrow y + (x_i - x) \sin \theta + (y_i - y) \cos \theta.\end{aligned}$$

The Python implementation in class `Curve`, assuming that  $\theta$  is given in degrees and not in radians, becomes

```
def rotate(self, angle, x=0, y=0):
    angle = angle*pi/180
    c = cos(angle); s = sin(angle)
    xnew = x + (self.x - x)*c - (self.y - y)*s
    ynew = y + (self.x - x)*s + (self.y - y)*c
    self.x = xnew
    self.y = ynew
```

The `rotate` method in class `Shape` is identical to the `draw`, `scale`, and `translate` methods except that we have other arguments:

```
class Shape:
    ....
    def rotate(self, angle, x=0, y=0):
        for shape in self.shapes:
            shape.rotate(angle, x, y)
```

*Application: Rolling Wheel.* To demonstrate the effect of translation and rotation we can roll a wheel on the screen. First we draw the wheel and rotate it a bit to demonstrate the basic operations:

```
center = (6,2) # the wheel's center point
w1 = Wheel(center=center, radius=2, inner_radius=0.5, nlines=7)
# rotate the wheel 2 degrees around its center point:
w1.rotate(angle=2, center[0], center[1])
w1.draw()
display()
```

Now we want to roll the wheel by making many such small rotations. At the same time we need to translate the wheel since rolling an arc length  $L = R\theta$ , where  $\theta$  is the rotation angle (in radians) and  $R$  is the outer radius of the wheel, implies that the center point moves a distance  $L$  to the left ( $\theta > 0$  means counter clockwise rotation). In code we must therefore combine rotation with translation:

```
L = radius*angle*pi/180 # translation = arc length
w1.rotate(angle, center[0], center[1])
w1.translate(-L, 0)
center = (center[0] - L, center[1])
```

We are now in a position to put the rotation and translation operations in a for loop and make a complete function:

```
def rolling_wheel(total_rotation_angle):
    """Animation of a rotating wheel."""
    set_coordinate_system(xmin=0, xmax=10, ymin=0, ymax=10)

    center = (6,2)
    radius = 2.0
    angle = 2.0
    w1 = Wheel(center=center, radius=radius,
               inner_radius=0.5, nlines=7)
    for i in range(int(total_rotation_angle/angle)):
        w1.draw()
        display()

        L = radius*angle*pi/180 # translation = arc length
        w1.rotate(angle, center[0], center[1])
        w1.translate(-L, 0)
        center = (center[0] - L, center[1])

    erase()
```

To control the visual “velocity” of the wheel, we can insert a pause between each frame in the for loop. A call to `time.sleep(s)`, where `s` is the length of the pause in seconds, can do this for us.

Another convenient feature is to save each frame drawn in the for loop as a hardcopy in PNG format and then, after the loop, make an animated GIF file based on the individual PNG frames. The latter operation is performed either by the `movie` function from `scitools.std` or by the `convert` program from the ImageMagick suite. With the latter you write the following command in a terminal window:

```
convert -delay 50 -loop 1000 xxx tmp_movie.gif
```

Here, `xxx` is a space-separated list of all the PNG files, and `tmp_movie.gif` is the name of the resulting animated GIF file. We can easily make `xxx` by collecting the names of the PNG files from the loop in a list object, and then join the names. The `convert` command can be run as an `os.system` call.

The complete `rolling_wheel` function, incorporating the mentioned movie making, will then be

```
def rolling_wheel(total_rotation_angle):
    """Animation of a rotating wheel."""
    set_coordinate_system(xmin=0, xmax=10, ymin=0, ymax=10)

    import time
    center = (6,2)
    radius = 2.0
    angle = 2.0
    pngfiles = []
    w1 = Wheel(center=center, radius=radius,
               inner_radius=0.5, nlines=7)
    for i in range(int(total_rotation_angle/angle)):
        w1.draw()
        display()

        filename = 'tmp_%03d' % i
        pngfiles.append(filename + '.png')
        hardcopy(filename)
        time.sleep(0.3) # pause 0.3 sec

        L = radius*angle*pi/180 # translation = arc length
        w1.rotate(angle, center[0], center[1])
        w1.translate(-L, 0)
        center = (center[0] - L, center[1])

        erase() # erase the screen before new figure

    cmd = 'convert -delay 50 -loop 1000 %s tmp_movie.gif' \
          % (' '.join(pngfiles))
    import commands
    failure, output = commands.getstatusoutput(cmd)
    if failure: print 'Could not run', cmd
```

The last two lines run a command, from Python, as we would run the command in a terminal window. The resulting animated GIF file can be viewed with `animate tmp_movie.gif` as a command in a terminal window.

## 9.6 Summary

### 9.6.1 Chapter Topics

A subclass inherits everything from its superclass, both attributes and methods. The subclass can add new attributes, overload methods, and thereby enrich or restrict functionality of the superclass.

*Subclass Example.* Consider class `Gravity` from Chapter 7.8.1 for representing the gravity force  $GMm/r^2$  between two masses  $m$  and  $M$

being a distance  $r$  apart. Suppose we want to make a class for the electric force between two charges  $q_1$  and  $q_2$ , being a distance  $r$  apart in a medium with permittivity  $\epsilon_0$  is  $Gq_1q_2/r^2$ , where  $G^{-1} = 4\pi\epsilon_0$ . We use the approximate value  $G = 8.99 \cdot 10^9 \text{ Nm}^2/\text{C}^2$  (C is the Coulumb unit used to measure electric charges such as  $q_1$  and  $q_2$ ). Since the electric force is similar to the gravity force, we can easily implement the electric force as a subclass of `Gravity`. The implementation just needs to redefine the value of  $G$ !

```
class CoulombsLaw(Gravity):
    def __init__(self, q1, q2):
        Gravity.__init__(self, q1, q2)
        self.G = 8.99E9
```

We can now call the inherited `force(r)` method to compute the electric force and the `visualize` method to make a plot of the force:

```
c = CoulombsLaw(1E-6, -2E-6)
print 'Electric force:', c.force(0.1)
c.visualize(0.01, 0.2)
```

However, the `plot` method inherited from class `Gravity` has an inappropriate title referring to “Gravity force” and the masses  $m$  and  $M$ . An easy fix could be to have the plot title as an attribute set in the constructor. The subclass can then override the contents of this attribute, as it overrides `self.G`. It is quite common to discover that a class needs adjustments if it is to be used as superclass.

*Subclassing in General.* The typical sketch of creating a subclass goes as follows:

```
class SuperClass:
    def __init__(self, p, q):
        self.p, self.q = p, q

    def where(self):
        print 'In superclass', self.__class__.__name__

    def compute(self, x):
        self.where()
        return self.p*x + self.q

class SubClass(SuperClass):
    def __init__(self, p, q, a):
        SuperClass.__init__(self, p, q)
        self.a = a

    def where(self):
        print 'In subclass', self.__class__.__name__

    def compute(self, x):
        self.where()
        return SuperClass.compute(self, x) + self.a*x**2
```

This example shows how a subclass extends a superclass with one attribute ( $a$ ). The subclass’ `compute` method calls the corresponding su-

perclass method, as well as the overloaded method `where`. Let us invoke the `compute` method through superclass and subclass instances:

```
>>> super = SuperClass(1, 2)
>>> sub = SubClass(1, 2, 3)
>>> v1 = super.compute(0)
In superclass SuperClass
>>> v2 = sub.compute(0)
In subclass SubClass
In subclass SubClass
```

Observe that in the subclass `sub`, method `compute` calls `self.where`, which translates to the `where` method in `SubClass`. Then the `compute` method in `SuperClass` is invoked, and this method also makes a `self.where` call, which is a call to `SubClass`' `where` method (think of what `self` is here, it is `sub`, so it is natural that we get `where` in the subclass (`sub.where`) and not `where` in the superclass part of `sub`).

In this example, classes `SuperClass` and `SubClass` constitute a class hierarchy. Class `SubClass` inherits the attributes `p` and `q` from its superclass, and overrides the methods `where` and `compute`.

### 9.6.2 Summarizing Example: Input Data Reader

The summarizing example of this chapter concerns a class hierarchy for simplifying reading input data into programs. Input data may come from several different sources: the command line, a file, or from a dialog with the user, either of `input` form or in a graphical user interface (GUI). Therefore it makes sense to create a class hierarchy where subclasses are specialized to read from different sources and where the common code is placed in a superclass. The resulting tool will make it easy for you to let your programs read from many different input sources by adding just a few lines.

*Problem.* Let us motivate the problem by a case where we want to write a program for dumping  $n$  function values of  $f(x)$  to a file for  $x \in [a, n]$ . The core part of the program typically reads

```
outfile = open(filename, 'w')
from numpy import linspace
for x in linspace(a, b, n):
    outfile.write('%12g %12g\n' % (x, f(x)))
outfile.close()
```

Our purpose is to read data into the variables `a`, `b`, `n`, `filename`, and `f`. For the latter we want to specify a formula and use the `StringFunction` tool (Chapter 3.1.4) to make the function `f`:

```
from scitools.StringFunction import StringFunction
f = StringFunction(formula)
```

How can we read `a`, `b`, `n`, `formula`, and `filename` conveniently into the program?

The basic idea is that we place the input data in a dictionary, and create a tool that can update this dictionary from sources like the command line, a file, a GUI, etc. Our dictionary is then

```
p = dict(formula='x+1', a=0, b=1, n=2, filename='tmp.dat')
```

This dictionary specifies the names of the input parameters to the program and the default values of these parameters.

Using the tool is a matter of feeding `p` into the constructor of a subclass in the tools' class hierarchy and extract the parameters into, for example, distinct variables:

```
inp = Subclassname(p)
a, b, filename, formula, n = inp.get_all()
```

Depending on what we write as `Subclassname`, the five variables can be read from the command line, the terminal window, a file, or a GUI. The task now is to implement a class hierarchy to facilitate the described flexible reading of input data.

*Solution.* We first create a very simple superclass `ReadInput`. Its main purpose is to store the parameter dictionary as an attribute, provide a method `get` to extract single values, and a method `get_all` to extract all parameters into distinct variables:

```
class ReadInput:
    def __init__(self, parameters):
        self.p = parameters

    def get(self, parameter_name):
        return self.p[parameter_name]

    def get_all(self):
        return [self.p[name] for name in sorted(self.p)]

    def __str__(self):
        import pprint
        return pprint.pformat(self.p)
```

Note that we in the `get_all` method must sort the keys in `self.p` such that the list of returned variables is well defined. In the calling program we can then list variables in the same order as the alphabetic order of the parameter names, for example:

```
a, b, filename, formula, n = inp.get_all()
```

The `__str__` method applies the `pprint` module to get a pretty print of all the parameter names and their values.

Class `ReadInput` cannot read from any source – subclasses are supposed to do this. The forthcoming text describes various types of subclasses for various types of reading input.

*Prompting the User.* The perhaps simplest way of getting data into a program is to use `raw_input`. We then prompt the user with a text `Give name:` and get an appropriate object back (recall that strings must be enclosed in quotes). The subclass `PromptUser` for doing this then reads

```
class PromptUser(ReadInput):
    def __init__(self, parameters):
        ReadInput.__init__(self, parameters)
        self._prompt_user()

    def _prompt_user(self):
        for name in self.p:
            self.p[name] = eval(raw_input("Give " + name + ": "))
```

Note the underscore in `_prompt_user`: the underscore signifies that this is a “private” method in the `PromptUser` class, not intended to be called by users of the class.

There is a major difficulty with using `eval` on the input from the user. When the input is intended to be a string object, such as a filename, say `tmp.inp`, the program will perform the operation `eval(tmp.inp)`, which leads to an exception because `tmp.inp` is treated as a variable `inp` in a module `tmp` and not as the string `'tmp.inp'`. To solve this problem, we use the `str2obj` function from the `scitools.misc` module. This function will return the right Python object also in the case where the argument should result in a string object (see Chapter 3.6.1 for some information about `str2obj`). The bottom line is that `str2obj` acts as a safer `eval(raw_input(...))` call. The key assignment in class `PromptUser` is then changed to

```
self.p[name] = str2obj(raw_input("Give " + name + ": "))
```

*Reading from File.* We can also place `name = value` commands in a file and load this information into the dictionary `self.p`. An example of a file can be

```
formula    = sin(x) + cos(x)
filename   = tmp.dat
a          = 0
b          = 1
```

In this example we have omitted `n`, so we rely on its default value.

A problem is how to give the filename. The easy way out of this problem is to read from standard input, and just redirect standard input from a file when we run the program. For example, if the filename is `tmp.inp`, we run the program as follows in a terminal window<sup>11</sup>

---

Terminal

---

```
Unix/DOS> python myprog.py < tmp.inp
```

---

<sup>11</sup> The redirection of standard input from a file does not work in IPython so we are in this case forced to run the program in a terminal window.



To interpret the contents of the file, we read line by line, split each line with respect to `=`, use the left-hand side as the parameter name and the right-hand side as the corresponding value. It is important to strip away unnecessary blanks in the name and value. The complete class now reads

```
class ReadInputFile(ReadInput):
    def __init__(self, parameters):
        ReadInput.__init__(self, parameters)
        self._read_file()

    def _read_file(self, infile=sys.stdin):
        for line in infile:
            if "=" in line:
                name, value = line.split("=")
                self.p[name.strip()] = str2obj(value.strip())
```

A nice feature with reading from standard input is that if we do not redirect standard input to a file, the program will prompt the user in the terminal window, where the user can give commands of the type `name = value` for setting selected input data. A `Ctrl-D` is needed to terminate the interactive session in the terminal window and continue execution of the program.

*Reading from the Command Line.* For input from the command line we assume that parameters and values are given as option-value pairs, e.g., as in

```
--a 1 --b 10 --n 101 --formula "sin(x) + cos(x)"
```

We apply the `getopt` module (Chapter 3.2.4) to parse the command-line arguments. The list of legal option names must be constructed from the list of keys in the `self.p` dictionary. The complete class takes the form

```
class ReadCommandLine(ReadInput):
    def __init__(self, parameters):
        self.sys_argv = sys.argv[1:] # copy
        ReadInput.__init__(self, parameters)
        self._read_command_line()

    def _read_command_line(self):
        # make getopt list of options:
        option_names = [name + "=" for name in self.p]
        try:
            options, args = getopt.getopt(self.sys_argv,
                                          '', option_names)
        except getopt.GetoptError, e:
            print 'Error in command-line option:\n', e
            sys.exit(1)

        for option, value in options:
            for name in self.p:
                if option == "--" + name:
                    self.p[name] = str2obj(value)
```

*Reading from a GUI.* We can with a little extra effort also make a graphical user interface (GUI) for reading the input data. An example of a user interface is displayed in Figure 9.14. Since the technicalities of the implementation is beyond the scope of this book, we do not show the subclass `GUI` that creates the GUI and loads the user input into the `self.p` dictionary.

a	0
formula	x+1
b	10
filename	tmp.dat
n	2
Run program	

**Fig. 9.14** Screen dump of a graphical user interface to read input data into a program (class `GUI` in the `ReadInput` hierarchy).

*More Flexibility in the Superclass.* Some extra flexibility can easily be added to the `get` method in the superclass. Say we want to extract a variable number of parameters:

```
a, b, n = inp.get('a', 'b', 'n') # 3 variables
n = inp.get('n')                 # 1 variable
```

The key to this extension is to use a variable number of arguments as explained in Appendix E.5.1:

```
class ReadInput:
    ...
    def get(self, *parameter_names):
        if len(parameter_names) == 1:
            return self.p[parameter_names[0]]
        else:
            return [self.p[name] for name in parameter_names]
```

*Demonstrating the Tool.* Let us show how we can use the classes in the `ReadInput` hierarchy. We apply the motivating example described earlier. The name of the program is `demo_ReadInput.py`. As first command-line argument it takes the name of the input source, given as the name of a subclass in the `ReadInput` hierarchy. The code for loading input data from any of the sources supported by the `ReadInput` hierarchy goes as follows:

```
p = dict(formula='x+1', a=0, b=1, n=2, filename='tmp.dat')
from ReadInput import *
input_reader = eval(sys.argv[1]) # PromptUser, ReadInputFile, ...
del sys.argv[1] # otherwise getopt does not work properly...
```

```
inp = input_reader(p)
a, b, filename, formula, n = inp.get_all()
print inp
```

Note how convenient `eval` is to automatically create the right subclass for reading input data.

Our first try on running this program applies the `PromptUser` class:

---

Terminal

---

```
demo_ReadInput.py PromptUser
Give a: 0
Give formula: sin(x) + cos(x)
Give b: 10
Give filename: function_data
Give n: 101
{'a': 0,
 'b': 10,
 'filename': 'function_data',
 'formula': 'sin(x) + cos(x)',
 'n': 101}
```

---

The next example reads data from a file `tmp.inp` with the same contents as shown under the *Reading from File* paragraph above<sup>12</sup>.

---

Terminal

---

```
demo_ReadInput.py ReadFileInput < tmp.inp
{'a': 0, 'b': 1, 'filename': 'tmp.dat',
 'formula': 'sin(x) + cos(x)', 'n': 2}
```

---

We can also drop the redirection of standard input to a file, and instead run an interactive session in IPython or the terminal window:

---

Terminal

---

```
demo_ReadInput.py ReadFileInput
n = 101
filename = myfunction_data_file.dat
^D
{'a': 0,
 'b': 1,
 'filename': 'myfunction_data_file.dat',
 'formula': 'x+1',
 'n': 101}
```

---

Note that `Ctrl-D` is needed to end the interactive session with the user and continue program execution.

Command-line arguments can also be specified:

---

Terminal

---

```
demo_ReadInput.py ReadCommandLine \
    --a -1 --b 1 --formula "sin(x) + cos(x)"
{'a': -1, 'b': 1, 'filename': 'tmp.dat',
 'formula': 'sin(x) + cos(x)', 'n': 2}
```

---

Finally, we can run the program with a GUI,

<sup>12</sup> This command with redirection from file must be run from a standard terminal window, not in an interactive IPython session.

---

Terminal

---

```
demo_ReadInput.py GUI
{'a': -1, 'b': 10, 'filename': 'tmp.dat',
 'formula': 'x+1', 'n': 2}
```

---

The GUI is shown in Figure 9.14.

Fortunately, it is now quite obvious how to apply the `ReadInput` hierarchy of classes in your own programs to simplify input. Especially in applications with a large number of parameters one can initially define these in a dictionary and then automatically create quite comprehensive user interfaces where the user can specify only some subset of the parameters (if the default values for the rest of the parameters are suitable).

## 9.7 Exercises

**Exercise 9.1.** *Demonstrate the magic of inheritance.*

Consider class `Line` from Chapter 9.1.1 and a subclass `Parabola0` defined as

```
class Parabola0(Line):
    pass
```

That is, class `Parabola0` does not have any own code, but it inherits from class `Line`. Demonstrate in a program or interactive session, using methods from Chapter 7.6.5, that an instance of class `Parabola0` contains everything (i.e., all attributes and methods) that an instance of class `Line` contains. Name of program file: `dir_subclass.py`. ◇

**Exercise 9.2.** *Inherit from classes in Ch. 9.1.*

The task in this exercise is to make a class `Cubic` for cubic functions

$$c_3x^3 + c_2x^2 + c_1x + c_0$$

with a call operator and a `table` method as in classes `Line` and `Parabola` from Chapter 9.1. Implement class `Cubic` by inheriting from class `Parabola`, and call up functionality in class `Parabola` in the same way as class `Parabola` calls up functionality in class `Line`.

Make a similar class `Poly4` for 4-th degree polynomials

$$c_4x^4 + c_3x^3 + c_2x^2 + c_1x + c_0$$

by inheriting from class `Cubic`. Insert `print` statements in all the `__call__` methods where you print out in which class you are. Evaluate cubic and a 4-th degree polynomial at a point, and observe the print-outs from all the superclasses. Name of program file: `Cubic_Poly4.py`.

◇

**Exercise 9.3.** *Inherit more from classes in Ch. 9.1.*

Implement a class for the function  $f(x) = A \sin(wx) + ax^2 + bx + c$ . The class should have a call operator for evaluating the function for some argument  $x$ , and a constructor that takes the function parameters  $A$ ,  $w$ ,  $a$ ,  $b$ , and  $c$  as arguments. Also a `table` method as in classes `Line` and `Parabola` should be present. Implement the class by deriving it from class `Parabola` and call up functionality already implemented in class `Parabola` whenever possible. Name of program file: `sin_plus_quadratic.py`.  $\diamond$

**Exercise 9.4.** *Reverse the class hierarchy from Ch. 9.1.*

Let class `Polynomial` from Chapter 7.3.7 be a superclass and implement class `Parabola` as a subclass. The constructor in class `Parabola` should take the three coefficients in the parabola as separate arguments. Try to reuse as much code as possible from the superclass in the subclass. Implement class `Line` as a subclass specialization of class `Parabola` (let the constructor take the two coefficients as two separate arguments).

Which class design do you prefer – class `Line` as a subclass of `Parabola` and `Polynomial`, or `Line` as a superclass with extensions in subclasses? Name of program file: `Polynomial_hier.py`.  $\diamond$

**Exercise 9.5.** *Super- and subclass for a point.*

A point  $(x, y)$  in the plane can be represented by a class:

```
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __str__(self):
        return '(%g, %g)' % (self.x, self.y)
```

We can extend the `Point` class to also contain the representation of the point in polar coordinates. To this end, create a subclass `PolarPoint` whose constructor takes the polar representation of a point,  $(r, \theta)$ , as arguments. Store  $r$  and  $\theta$  as attributes and call the superclass constructor with the corresponding  $x$  and  $y$  values (recall the relations  $x = r \cos \theta$  and  $y = r \sin \theta$  between Cartesian and polar coordinates). Also, in class `PolarPoint`, add a `__str__` method which prints out  $r$ ,  $\theta$ ,  $x$ , and  $y$  of a point. Verify the implementation by initializing three points and printing these points. Name of program file: `PolarPoint.py`.  $\diamond$

**Exercise 9.6.** *Modify a function class by subclassing.*

Consider the `VelocityProfile` class from page 346 for computing the function  $v(r; \beta, \mu_0, n, R)$  in formula (4.20) on page 230. Suppose we want to have  $v$  explicitly as a function of  $r$  and  $n$  (this is necessary if we want to illustrate how the velocity profile, the  $v(r)$  curve, varies as  $n$  varies). We would then like to have a class `VelocityProfile2` that

is initialized with  $\beta$ ,  $\mu_0$ , and  $R$ , and that takes  $r$  and  $n$  as arguments in the `__call__` method. Implement such a class by inheriting from class `VelocityProfile` and by calling the `__init__` and `value` methods in the superclass. It should be possible to try the class out with the following statements:

```
v = VelocityProfile2(beta=0.06, mu0=0.02, R=2)
# evaluate v for various n values at r=0:
for n in 0.1, 0.2, 1:
    print v(0, n)
```

Name of program file: `VelocityProfile2.py`.  $\diamond$

**Exercise 9.7.** *Explore the accuracy of difference formulas.*

The purpose of this exercise is to investigate the accuracy of the `Backward1`, `Forward1`, `Forward3`, `Central2`, `Central4`, `Central6` methods for the function<sup>13</sup>

$$v(x) = \frac{1 - e^{x/\mu}}{1 - e^{1/\mu}}.$$

To solve the exercise, modify the `src/oo/Diff2_examples.py` program which produces tables of errors of difference approximations as discussed at the end of Chapter 9.2.3. Test the approximation methods for  $x = 0, 0.9$  and  $\mu = 1, 0.01$ . Plot the  $v(x)$  function for the two  $\mu$  values using 1001 points. Name of program file: `boundary_layer_derivative.py`.  $\diamond$

**Exercise 9.8.** *Implement a subclass.*

Make a subclass `Sine1` of class `FuncWithDerivatives` from Chapter 9.1.7 for the  $\sin x$  function. Implement the function only, and rely on the inherited `df` and `ddf` methods for computing the derivatives. Make another subclass `Sine2` for  $\sin x$  where you also implement the `df` and `ddf` methods using analytical expressions for the derivatives. Compare `Sine1` and `Sine2` for computing the first- and second-order derivatives of  $\sin x$  at two  $x$  points. Name of program file: `Sine12.py`.  $\diamond$

**Exercise 9.9.** *Make classes for numerical differentiation.*

Carry out Exercise 7.15. Find the common code in the classes `Derivative`, `Backward`, and `Central`. Move this code to a superclass, and let the three mentioned classes be subclasses of this superclass. Compare the resulting code with the hierarchy shown in Chapter 9.2.1. Name of program file: `numdiff_classes.py`.  $\diamond$

**Exercise 9.10.** *Implement a new subclass for differentiation.*

A one-sided, three-point, second-order accurate formula for differentiating a function  $f(x)$  has the form

$$f'(x) \approx \frac{f(x - 2h) - 4f(x - h) + 3f(x)}{2h}. \quad (9.47)$$

<sup>13</sup> This function is discussed more in detail in Exercise 4.26.

Implement this formula in a subclass `Backward2` of class `Diff` from Chapter 9.2. Compare `Backward2` with `Backward1` for  $g(t) = e^{-t}$  for  $t = 0$  and  $h = 2^{-k}$  for  $k = 0, 1, \dots, 14$  (write out the errors in  $g'(t)$ ). Name of program file: `Backward2.py`.  $\diamond$

**Exercise 9.11.** *Understand if a class can be used recursively.*

Suppose you want to compute  $f''(x)$  of some mathematical function  $f(x)$ , and that you apply class `Diff3` from Chapter 9.2.6 twice:

```
ddf = Diff3(Diff3(f, 'central', 2), 'central', 2)
```

Will this work? Hint: Follow the program flow, and find out what the resulting formula will be. Then see if this formula coincides with a formula you know for approximating  $f''(x)$ .  $\diamond$

**Exercise 9.12.** *Represent people by a class hierarchy.*

Classes are often used to model objects in the real world. We may represent the data about a person in a program by a class `Person`, containing the person's name, address, phone number, date of birth, and nationality. A method `__str__` may print the person's data. Implement such a class `Person`.

A worker is a person with a job. In a program, a worker is naturally represented as class `Worker` derived from class `Person`, because a worker *is* a person, i.e., we have an is-a relationship. Class `Worker` extends class `Person` with additional data, say name of company, company address, and job phone number. The print functionality must be modified accordingly. Implement this `Worker` class.

A scientist is a special kind of a worker. Class `Scientist` may therefore be derived from class `Worker`. Add data about the scientific discipline (physics, chemistry, mathematics, computer science, ...). One may also add the type of scientist: theoretical, experimental, or computational. The value of such a type attribute should not be restricted to just one category, since a scientist may be classified as, e.g., both experimental and computational (i.e., you can represent the value as a list or tuple). Implement class `Scientist`.

Researcher, postdoc, and professor are special cases of a scientist. One can either create classes for these job positions, or one may add an attribute (`position`) for this information in class `Scientist`. We adopt the former strategy. When, e.g., a researcher is represented by a class `Researcher`, no extra data or methods are needed. In Python we can create such an “empty” class by writing `pass` (the empty statement) as the class body:

```
class Researcher(Scientist):
    pass
```

Finally, make a demo program where you create and print instances of classes `Person`, `Worker`, `Scientist`, `Researcher`, `Postdoc`, and `Professor`. Print out the attribute contents of each instance (use the `dir` function).

*Remark.* An alternative design is to introduce a class `Teacher` as a special case of `Worker` and let `Professor` be both a `Teacher` and `Scientist`, which is natural. This implies that class `Professor` has two superclasses, `Teacher` and `Scientist`, or equivalently, class `Professor` inherits from two superclasses. This is known as *multiple inheritance* and technically achieved as follows in Python:

```
class Professor(Teacher, Scientist):
    pass
```

It is a continuous debate in computer science whether multiple inheritance is a good idea or not. One obvious problem<sup>14</sup> in the present example is that class `Professor` inherits two names, one via `Teacher` and one via `Scientist` (both these classes inherit from `Person`). Neither of the two widely used languages Java and C# allow multiple inheritance. Nor in this book will we pursue the idea of multiple inheritance further. Name of program file: `Person.py`. ◇

**Exercise 9.13.** *Add a new class in a class hierarchy.*

Add the Monte Carlo integration method from Chapter 8.5.1 as a subclass in the `Integrator` hierarchy explained in Chapter 9.3. Import the superclass `Integrator` from the `integrate` module in the file with the new integration class. Test the Monte Carlo integration class in a case with known analytical solution. Name of program file: `MCint_class.py`. ◇

**Exercise 9.14.** *Change the user interface of a class hierarchy.*

All the classes in the `Integrator` hierarchy from Chapter 9.3 take the integration limits  $a$  and  $b$  plus the number of integration points  $n$  as input to the constructor. The `integrate` method takes the function to integrate,  $f(x)$ , as parameter. Another possibility is to feed  $f(x)$  to the constructor and let `integrate` take  $a$ ,  $b$ , and  $n$  as parameters. Make this change to the `integrate.py` file with the `Integrator` hierarchy. Name of program file: `integrate2.py`. ◇

**Exercise 9.15.** *Compute convergence rates of numerical integration methods.*

Most numerical methods have a discretization parameter, call it  $n$ , such that if  $n$  increases (or decreases), the method performs better. Often, the relation between the error in the numerical approximation (compared with the exact analytical result) can be written as

$$E = Cn^r,$$

where  $E$  is the error, and  $C$  and  $r$  are constants.

<sup>14</sup> It is usually not a technical problem, but more a conceptual problem when the world is modeled by objects in a program.



Suppose you have performed an experiment with a numerical method using discretization parameters  $n_0, n_1, \dots, n_N$ . You have computed the corresponding errors  $E_0, E_1, \dots, E_N$  in a test problem with an analytical solution. One way to estimate  $r$  goes as follows. For two successive experiments we have

$$E_{i-1} = Cn_{i-1}^r$$

and

$$E_i = Cn_i^r.$$

Divide the first equation by the second to eliminate  $C$ , and then take the logarithm to solve for  $r$ :

$$r = \frac{\ln(E_{i-1}/E_i)}{\ln(n_{i-1}/n_i)}.$$

We can compute  $r$  for all pairs of two successive experiments. Usually, the “last  $r$ ”, corresponding to  $i = N$  in the formula above, is the “best”  $r$  value<sup>15</sup>. Knowing this  $r$ , we can compute  $C$  as  $E_N n_N^{-r}$ .

Having stored the  $n_i$  and  $E_i$  values in two lists `n` and `E`, the following code snippet computes  $r$  and  $C$ :

```
from scitools.convergenrate import convergence_rate
C, r = convergence_rate(n, E)
```

Construct a test problem for integration where you know the analytical result of the integral. Run different numerical methods (the midpoint method, the Trapezoidal method, Simpson’s method, Monte Carlo integration) with the number of evaluation points  $n = 2^k + 1$  for  $k = 2, \dots, 11$ , compute corresponding errors, and use the code snippet above to compute the  $r$  value for the different methods in questions. The higher the absolute error of  $r$  is, the faster the method converges to the exact result as  $n$  increases, and the better the method is. Which is the best and which is the worst method?

Let the program file import methods from the `integrate` module and the module with the Monte Carlo integration method from Exercise 9.13. Name of program file: `integrators_convergence.py`. ◇

**Exercise 9.16.** *Add common functionality in a class hierarchy.*

Suppose you want to use classes in the `Integrator` hierarchy from Chapter 9.3 to calculate integrals of the form

$$F(x) = \int_a^x f(t)dt.$$

<sup>15</sup> This guideline is rough. If the method convergences, and round-off errors do not influence the values of  $E_i$ , the guideline is good. However, for very large/small  $n$  round-off errors can cause the method to diverge, and then the “last  $r$ ” is not a relevant value to pick.

Such functions  $F(x)$  can be efficiently computed by the method from Exercise 7.22. Implement this computation of  $F(x)$  in an additional method in the superclass `Integrator`. Test that the implementation is correct for  $f(x) = 2x - 3$  for all the implemented integration methods (the Midpoint, Trapezoidal and Gauss-Legendre methods, as well as Simpson's rule, integrate a linear function exactly). Name of program file: `integrate_efficient.py`.  $\diamond$

**Exercise 9.17.** *Make a class hierarchy for root finding.*

Given a general nonlinear equation  $f(x) = 0$ , we want to implement classes for solving such an equation, and organize the classes in a class hierarchy. Make classes for three methods: Newton's method (Chapter 5.1.9), the Bisection method (Chapter 3.6.2), and the Secant method (Exercise 5.14). Move common code (starting values, the  $f(x)$  functions, parameters in termination criteria, etc.) to a common superclass. Do Exercise 5.15 using the new class hierarchy. Name of program file: `Rootfinders.py`.  $\diamond$

**Exercise 9.18.** *Use the ODESolver hierarchy to solve a simple ODE.*

Solve the ODE problem  $u' = u/2$  with  $u(0) = 1$ , using a class in the `ODESolver` hierarchy. Choose  $\Delta t = 0.5$  and perform  $N = 12$  steps. Write out the approximate  $u_N$  together with the exact value  $e^{N\Delta t/2}$ . Name of program file: `ODESolver_demo.py`.  $\diamond$

**Exercise 9.19.** *Use the 4th-order Runge-Kutta on (B.34).*

Investigate if the 4th-order Runge-Kutta method is better than the Forward Euler scheme for solving the challenging ODE problem (B.34) from Exercise B.3 on page 621. Name of program file: `yx_ODE2.py`.  $\diamond$

**Exercise 9.20.** *Solve an ODE until constant solution.*

Newton's law of cooling,

$$\frac{dT}{dt} = -h(T - T_s) \quad (9.48)$$

can be used to see how the temperature  $T$  of an object changes because of heat exchange with the surroundings, which have a temperature  $T_s$ . The parameter  $h$ , with unit  $\text{s}^{-1}$  is an experimental constant (heat transfer coefficient) telling how efficient the heat exchange with the surroundings is. For example, (9.48) may model the cooling of a hot pizza taken out of the oven. The problem with applying (9.48), nevertheless, is that  $h$  must be measured. Suppose we have measured  $T$  at  $t = 0$  and  $t_1$ . We can use a rough Forward Euler approximation of (9.48) with one time step of length  $t_1$ ,

$$\frac{T(t_1) - T(0)}{t_1} = -h(T(0) - T_s),$$

to make the estimate

$$h = \frac{T(t_1) - T(0)}{t_1(T_s - T(0))}. \quad (9.49)$$

Suppose now you take a hot pizza out of the oven. The temperature of the pizza is 200 C at  $t = 0$  and 180 C after 20 seconds, in a room with temperature 20 C. Find an estimate of  $h$  from the formula above.

Solve (9.48) to find the evolution of the temperature of the pizza. Use class `ForwardEuler` or `RungeKutta4`, and supply a `terminate` function to the `solve` method so that the simulation stops when  $T$  is sufficiently close to the final room temperature  $T_s$ . Plot the solution. Name of program file: `pizza_cooling1.py`.  $\diamond$

**Exercise 9.21.** *Use classes in Exer. 9.20.*

Solve Exercise 9.20 with a class `Problem` containing the parameters  $h$ ,  $T_s$ ,  $T(0)$ ,  $t_1$ , and  $T(t_1)$  as attributes. The class should have a method `estimate_h` for returning an estimate of  $h$ , given the other parameters. Also a method `__call__` for computing the right-hand side must be included. The `terminate` function can be a method in the class as well. By using class `Problem`, we avoid having the physical parameters as global variables in the program. Name of program file: `pizza_cooling2.py`.  $\diamond$

**Exercise 9.22.** *Scale away parameters in Exer. 9.20.*

Use the scaling approach from Chapter 9.4.7 to “scale away” the parameters in the ODE in Exercise 9.20. That is, introduce a new unknown  $u = (T - T_s)/(T(0) - T_s)$  and a new time scale  $\tau = th$ . Find the ODE and the initial condition that governs the  $u(\tau)$  function. Make a program that computes  $u(\tau)$  until  $|u| < 0.001$ . Store the discrete  $u$  and  $\tau$  values in a file `u_tau.dat` if that file is not already present (you can use `os.path.isfile(f)` to test if a file with name `f` exists). Create a function `T(u, tau, h, T0, Ts)` that loads the  $u$  and  $\tau$  data from the `u_tau.dat` file and returns two arrays with  $T$  and  $t$  values, corresponding to the computed arrays for  $u$  and  $\tau$ . Plot  $T$  versus  $t$ . Give the parameters  $h$ ,  $T_s$ , and  $T(0)$  on the command line. Note that this program is supposed to solve the ODE once and then recover any  $T(t)$  solution by a simple scaling of the single  $u(\tau)$  solution. Name of program file: `pizza_cooling3.py`.  $\diamond$

**Exercise 9.23.** *Compare ODE methods.*

Equation (7.6) is a relevant model for radioactive decay. The function  $u(t)$  is the fraction of particles that remains in the radioactive substance at time  $t$ . The parameter  $a$  is the inverse of the so-called mean lifetime of the substance. The initial condition is  $u(0) = 1$ .

Introduce a class `Decay` to hold information about the physical problem: the parameter  $a$  and a `__call__` method for computing the right-hand side  $-au$  of the ODE (see Chapters 7.4.4 or 9.4.8 for examples). Initialize an instance of class `Decay` with  $a = \ln(2)/5600$  1/years (this value of  $a$  corresponds to the Carbon-14 radioactive isotope whose

decay is used extensively in dating organic material that is tens of thousands of years old).

Solve (7.6) by both the Forward Euler and the 4-th order Runge-Kutta method, using the `ForwardEuler` and the `RungeKutta4` classes in the `ODESolver` hierarchy. Use a time step of 500 years, and simulate decay for 20,000 years (let the time unit be 1 year). Plot the two solutions. Write out the final  $N$  value from the simulations and compare it with the exact solution  $N(t) = N_0 e^{-N\Delta t/\tau}$ . Name of program file: `radioactive_decay.py`.  $\diamond$

**Exercise 9.24.** *Solve two coupled ODEs for radioactive decay.*

Consider two radioactive substances A and B. The nuclei in substance A decay to form nuclei of type B with a mean lifetime  $\tau_A$ , while substance B decay to form type A nuclei with a mean lifetime  $\tau_B$ . Letting  $u_A$  and  $u_B$  be the fractions of the initial amount of material in substance A and B, respectively, the following system of ODEs governs the evolution of  $u_A(t)$  and  $u_B(t)$ :

$$u'_A = u_B/\tau_B - u_A/\tau_A, \quad (9.50)$$

$$u'_B = u_A/\tau_A - u_B/\tau_B, \quad (9.51)$$

with  $u_A(0) = u_B(0) = 1$ . As in Exercise 9.23, introduce a problem class, which holds the parameters  $\tau_A$  and  $\tau_B$  and offers a `__call__` method to compute the right-hand side vector of the ODE system, i.e.,  $(u_B/\tau_B - u_A/\tau_A, u_A/\tau_A - u_B/\tau_B)$ . Solve for  $u_A$  and  $u_B$  using a subclass in the `ODESolver` hierarchy and the parameter choice  $\tau_A = 8$  minutes,  $\tau_B = 40$  minutes, and  $\Delta t = 10$  seconds. Plot  $u_A$  and  $u_B$  against time measured in minutes. From the ODE system it follows that the ratio  $u_A/u_B \rightarrow \tau_A/\tau_B$  as  $t \rightarrow \infty$  (assuming  $u'_A = u'_B = 0$  in the limit  $t \rightarrow \infty$ ). Check that the solutions fulfill this requirement (this is a partial verification of the program). Name of program file: `radioactive_decay2.py`.  $\diamond$

**Exercise 9.25.** *Compare methods for solving the ODE (B.36).*

Consider emptying a tank of water as described in Exercise B.7 on page 554. The governing ODE problem is (B.36) with  $h(0) = h_0$ . Make a class `Tank` for storing the physical parameters of the problem:  $r$ ,  $R$ ,  $g$ , and  $h_0$ . This class should also have a `__call__` method for defining the right-hand side of (B.36). Solve this ODE problem using the `ForwardEuler`, `BackwardEuler`, and `RungeKutta4` classes in the `ODESolver` hierarchy. Apply data in the `Tank` instance to initialize the solver classes. Read  $\Delta t$  from the command line and try out values between 5 and 50 s. Compare the numerical solutions in a plot. Comment upon the quality of the various methods to compute a correct limiting value of  $h(0)$  as  $\Delta t$  is varied. Name of program file: `tank_ODE_3methods.py`.  $\diamond$

**Exercise 9.26.** *Code a 2nd-order Runge-Kutta method; function.*

Implement the 2nd-order Runge-Kutta method specified in formula (9.25) for solving ordinary differential equations. Use a plain function `RungeKutta2` of the type shown in Chapter 7.4.1 for the Forward Euler method. Construct a test problem where you know the analytical solution, and plot the difference between the numerical and analytical solution. Name of program file: `RungeKutta2_func.py`. ◇

**Exercise 9.27.** *Code a 2nd-order Runge-Kutta method; class.*

Make a new subclass `RungeKutta2` in the `ODESolver` hierarchy from Chapter 9.4 for solving ordinary differential equations with the 2nd-order Runge-Kutta method specified in formula (9.25). Construct a test problem where you know the analytical solution, and plot the difference between the numerical and analytical solution. Store the `RungeKutta2` class and the test problem in a file where the base class `ODESolver` is imported from the `ODESolver` module. Name of program file: `RungeKutta2.py`. ◇

**Exercise 9.28.** *Implement a midpoint method for ODEs.*

This exercise is similar to Exercise 9.27, but the purpose now is to implement the midpoint method specified in formula (9.24).

Compare in a plot the midpoint method with the Forward Euler and 4th-order Runge-Kutta methods and the exact solution for the problem  $u' = u$ ,  $u(0) = 1$ , with  $\Delta t = 0.5$  and 10 steps. Name of program file: `Midpoint.py`. ◇

**Exercise 9.29.** *Implement a modified Euler method for ODEs.*

Do Exercise 7.29 and incorporate the class in the `ODESolver` hierarchy. Compare the method with other methods in the same test problem as in Exercise 9.28. Name of program file: `ModifiedEuler.py`. ◇

**Exercise 9.30.** *Improve the implementation in Exer. 7.25.*

We consider the physical problem of an object falling or rising in a fluid as described in Exercise 7.25. The purpose now is to solve the governing ODE (7.16) using classes in the `ODESolver` hierarchy. We also want to set the physical and numerical parameters of the problem on the command line.

First make a class for defining the right-hand side of (7.16). The physical parameters needed in the definition of the right-hand side should be attributes in the class.

Continue with making a function `solve(method, f, v0, T, dt)` for solving (7.16). The argument `method` is the name of a subclass in the `ODESolver` hierarchy, `f` is the object defining the right-hand side of the ODE, `v0` is the initial condition, `T` is the final time for the simulation, and `dt` is the time step. The `solve` should return two arrays, one with the velocity values and one with the corresponding time values.

A separate function `read_input` can read all the input data from the command line, preferably using the `getopt` module, or better, the

`ReadInput` class hierarchy from Chapter 9.6.2. The `read_input` function first sets some default values, then reads input, and finally returns the set of variables that must be sent further to the `solve` function (see `src/box_spring/box_spring.py` for a similar example using `getopt`).

Implement the simple verification test as a function `verify`. The two other real test cases can be implemented in two separate functions, `parachute_jumper` and `rising_ball`. Let these three functions return the computed velocities and corresponding time points. Collect the right-hand side class and all the functions in a module file. Run one of the three cases from the test block, using a command-line argument to determine which case. Name of program file: `body_in_fluid2.py`. ◇

**Exercise 9.31.** *Visualize the different forces in Exer. 9.30.*

The purpose of this exercise is to plot the forces  $F_g$ ,  $F_b$ , and  $F_d$  in the model from Exercise 9.30 as functions of  $t$ . Seeing the relative importance of the forces as time develops gives an increased understanding of how the different forces contribute to change the velocity.

Import the functions from the module developed in Exercise 9.30 in a new program, call the function for computing one of the two real cases from Exercise 9.30, and feed the returned  $v$  to a new function for computing the forces  $F_g$ ,  $F_b$ , and  $F_d$ . Plot these three forces against time. Name of program file: `body_in_fluid_forces.py`. ◇

**Exercise 9.32.** *Find the body's position in Exer. 9.30.*

In Exercise 9.30 we compute  $v(t)$ . The position of the body,  $y(t)$ , is related to the velocity  $v$  by  $y'(t) = v(t)$ . Extend the program from Exercise 9.30 to solve the system

$$\begin{aligned}\frac{dy}{dt} &= v, \\ \frac{dv}{dt} &= -g \left(1 - \frac{\rho}{\rho_b}\right) - \frac{1}{2} C_D \frac{\rho A}{\rho_b V} |v|v.\end{aligned}$$

Name of program file: `body_in_fluid2.py`. ◇

**Exercise 9.33.** *Compare methods for solving (B.37)–(B.38).*

Consider the system of ODEs in Exercise B.8 for simulating an electric circuit. The purpose now is to compare the Forward Euler scheme with the 4-th order Runge-Kutta method. Make a class `Circuit` for storing the physical parameters of the problem ( $L$ ,  $R$ ,  $C$ ,  $E(t)$ ) as well as the initial conditions ( $I(0)$ ,  $Q(0)$ ). Class `Circuit` should also define the right-hand side of the ODE through a `__call__` method. Create two solver instances, one from the `ForwardEuler` class and one from the `RungeKutta4` class. Solve the ODE system using both methods. Plot the two  $I(t)$  solutions for comparison. As you will see, the Forward Euler scheme overestimates the amplitudes significantly, compared with the more accurate 4th-order Runge-Kutta method. Name of program file: `electric_circuit2.py`. ◇

**Exercise 9.34.** *Add the effect of air resistance on a ball.*

The differential equations governing the horizontal and vertical motion of a ball subject to gravity and air resistance read<sup>16</sup>

$$\frac{d^2x}{dt^2} = -\frac{3}{8}C_D\bar{\rho}a^{-1}\sqrt{\left(\frac{dx}{dt}\right)^2 + \left(\frac{dy}{dt}\right)^2}\frac{dx}{dt}, \quad (9.52)$$

$$\frac{d^2y}{dt^2} = -g - \frac{3}{8}C_D\bar{\rho}a^{-1}\sqrt{\left(\frac{dx}{dt}\right)^2 + \left(\frac{dy}{dt}\right)^2}\frac{dy}{dt}, \quad (9.53)$$

where  $(x, y)$  is the position of the ball ( $x$  is a horizontal measure and  $y$  is a vertical measure),  $g$  is the acceleration of gravity,  $C_D = 0.2$  is a drag coefficient,  $\bar{\rho}$  is the ratio of the density of air and the ball, and  $a$  is the radius of the ball. The latter two quantities can be taken as 0.017 and 11 cm for a football.

Let the initial condition be  $x = y = 0$  (start position in origo) and

$$dx/dt = v_0 \cos \theta, \quad dy/dt = v_0 \sin \theta,$$

where  $v_0$  is the magnitude of the initial velocity and  $\theta$  is the angle the velocity makes with the horizontal. For a hard football kick we can set  $v_0 = 120$  km/h and take  $\theta$  as 30 degrees.

Express the two second-order equations above as a system of four first-order equations with four initial conditions. Implement the right-hand side in a problem class where the physical parameters  $C_D$ ,  $\bar{\rho}$ ,  $a$ ,  $v_0$ , and  $\theta$  are stored along with the initial conditions.

Solve the ODE system for  $C_D = 0$  (no air resistance) and  $C_D = 0.2$ , and plot  $y$  as a function of  $x$  in both cases to illustrate the effect of air resistance. Use the 4-th order Runge-Kutta method. (Make sure you express all units in kg, m, s, and radians.) Name of program file: `kick2D.py`.  $\diamond$

**Exercise 9.35.** *Make a class for drawing an arrow.*

Make a class in the `Shape` hierarchy from Chapter 9.5 for drawing an arrow. An arrow consists of three lines, so the arrow class will naturally contain three `Line` instances. Let each line in the arrow head make an angle of 30 degrees with the main line, and let the length of the arrow head be 1/8 of the length of the main line. It is easiest to always draw a vertical arrow in the `Arrow` class. The constructor can then take a bottom point and the length of the arrow. With the `rotate` method the user can later rotate the arrow.

Make some arrows of different lengths, and call `rotate` to rotate them differently. Name of program file: `Arrow.py`.  $\diamond$

<sup>16</sup> The equations arise by combining the models in Exercises 1.10 and 1.14.

**Exercise 9.36.** *Make a class for drawing a person.*

A very simple sketch of a human being can be made of a circle for the head, two lines for the arms, one vertical line or a rectangle for the torso, and two lines for the legs. Make a class in the `Shape` hierarchy from Chapter 9.5 for drawing such a simple sketch of a person. Build the figure from `Line` and `Circle` instances. Supply the following arguments to the constructor: the center point of the head and the radius  $R$  of the head. Let the arms and the torso be of length  $4R$ , and the legs of length  $6R$ . The angle between the legs can be fixed (say 30 degrees), while the angle of the arms relative to the torso can be an argument to the constructor with a suitable default value. Name of program file: `draw_person.py`.  $\diamond$

**Exercise 9.37.** *Animate a person with waving hands.*

Make a subclass of the class from Exercise 9.36 where the constructor can take an argument describing the angle between the arms and the torso. Use this new class to animate a person who waves her/his hands. Name of program file: `draw_waving_person.py`.  $\diamond$

**Exercise 9.38.** *Make a class for drawing a car.*

A two-dimensional car can be drawn by putting together a rectangle, circles, arcs, and lines. Make a class in the `Shape` hierarchy from Chapter 9.5 for drawing a car, following the same principle as in Exercise 9.36. The constructor takes a length  $L$  of the car and the coordinates of a point  $p$ . The various shapes that build up the car must have dimensions relative to  $L$  and placements relative to  $p$ . Draw a small car and a large car in the same figure. Name of program file: `draw_car.py`.  $\diamond$

**Exercise 9.39.** *Make a car roll.*

Use the class for drawing a car in Exercise 9.38 and the ideas from Chapter 9.5.5 to make an animation of a rolling car. Implement the rolling functionality in a subclass of the car class from Exercise 9.38. Name of program file: `rolling_car.py`.  $\diamond$

**Exercise 9.40.** *Make a class for differentiating noisy data.*

Suppose you have some time series signal  $y(t_k)$  for  $k = 0, \dots, n - 1$ , where  $t_k = k\Delta t$  are time points. Differentiating such a signal can give very inaccurate results if the signal contains noise. Exercises 8.44–8.47 explore this topic, and Exercise 8.47 suggests to filter the signal. The purpose of the present exercise is to make a tool for differentiating noisy signals.

Make a class `DiffNoisySignal` where the constructor takes three arguments: the signal  $y(t_k)$  (as an array), the order of the desired derivative (as an `int`, either 1 or 2), and the name of the signal (as a string). A method `filter(self, n)` runs the filter from Exercise 8.47  $n$  times on the signal. The method `diff(self)` performs the differentiation and



stores the differentiated signal as an attribute in the class. There should also be some plotting methods: `plot(self)` for plotting the current (original or filtered) signal, `plot_diff(self)` for plotting the differentiated signal, `animate_filter` for animating the effect of filtering (run `filter` once per frame in the movie), and `animate_diff` for animating the evolution of the derivative when `filter` and `diff` are called once each per frame.

Implement the class and test it on the noisy signal

$$y(t_k) = \cos(2\pi t_k) + 0.1r_k, \quad t_k = k\Delta t, \quad k = 0, \dots, n-1,$$

with  $\Delta t = 1/60$ . The quantities  $r_k$  are random numbers in  $[0, 1)$ . Make animations with the `animate_filter` and `animate_diff` methods. Name of program file: `DiffNoisySignal.py`.  $\diamond$

**Exercise 9.41.** Find local and global extrema of a function.

Extreme points of a function  $f(x)$  are normally found by solving  $f'(x) = 0$ . A much simpler method is to evaluate  $f(x)$  for a set of discrete points in the interval  $[a, b]$  and look for local minima and maxima among these points. We work with  $n$  equally spaced points  $a = x_0 < x_1 < \dots < x_{n-1} = b$ ,  $x_i = a + ih$ ,  $h = (b - a)/(n - 1)$ .

1. First we find all local extreme points in the interior of the domain. Local minima are recognized by

$$f(x_{i-1}) > f(x_i) < f(x_{i+1}), \quad i = 1, \dots, n-2.$$

Similarly, at a local maximum point  $x_i$  we have

$$f(x_{i-1}) < f(x_i) > f(x_{i+1}), \quad i = 1, \dots, n-2.$$

We let  $P_{\min}$  be the set of  $x$  values for local minima and  $F_{\min}$  the set of the corresponding  $f(x)$  values at these minimum points. Two sets  $P_{\max}$  and  $F_{\max}$  are defined correspondingly, containing the maximum points and their values.

2. The boundary points  $x = a$  and  $x = b$  are for algorithmic simplicity also defined as local extreme points:  $x = a$  is a local minimum if  $f(a) < f(x_1)$ , and a local maximum otherwise. Similarly,  $x = b$  is a local minimum if  $f(b) < f(x_{n-2})$ , and a local maximum otherwise. The end points  $a$  and  $b$  and the corresponding function values must be added to the sets  $P_{\min}, P_{\max}, F_{\min}, F_{\max}$ .
3. The global maximum point is defined as the  $x$  value corresponding to the maximum value in  $F_{\max}$ . The global minimum point is the  $x$  value corresponding to the minimum value in  $F_{\min}$ .

Make a class `MinMax` with the following functionality:

- The constructor takes  $f(x)$ ,  $a$ ,  $b$ , and  $n$  as arguments, and calls a method `_find_extrema` to compute the local and global extreme points.
- The method `_find_extrema` implements the algorithm above for finding local and global extreme points, and stores the sets  $P_{\min}, P_{\max}, F_{\min}, F_{\max}$  as list attributes in the (`self`) instance.
- The method `get_global_minimum` returns the global minimum point ( $x$ ).
- The method `get_global_maximum` returns the global maximum point ( $x$ ).
- The method `get_all_minima` returns a list or array of all minimum points.
- The method `get_all_maxima` returns a list or array of all maximum points.
- The method `__str__` returns a string where all the min/max points are listed, plus the global extreme points.

Here is a sample code using class `MinMax`:

```
def f(x):
    return x**2*exp(-0.2*x)*sin(2*pi*x)

m = MinMax(f, 0, 4)
print m
```

The output becomes

```
All minima: 0.8056, 1.7736, 2.7632, 3.7584, 0
All maxima: 0.3616, 1.284, 2.2672, 3.2608, 4
Global minimum: 3.7584
Global maximum: 3.2608
```

Make sure that the program also works for functions without local extrema, e.g., linear functions  $f(x) = px + q$ . Name of program file: `minmaxf.py`.  $\diamond$

#### **Exercise 9.42.** *Improve the accuracy in Exer. 9.41.*

The algorithm in Exercise 9.41 finds a local extreme point if the function value at  $x_i$  is larger or smaller than the function values at the neighboring points. If we have found a minimum at  $x_i$ , all we know is that  $f(x)$  has a minimum in the interval  $(x_{i-1}, x_{i+1})$ . With  $h$  as the distance between the points, the error in the coordinate of an extreme point can be as high as  $2h$ . Of course, increasing the number of points decreases this error. Nevertheless, we may think of a computationally more efficient method, namely a bisection method for finding  $f'(x) = 0$  in  $(x_{i-1}, x_{i+1})$ . In class `MinMax`, add a method `_refine_extrema`, which goes through all the interior local minima and maxima, makes a callable object for  $f'(x)$  using the `Central2` class in the `Diff` hierarchy, and calls a bisection method to find where  $f'(x) = 0$ . The tolerance in the termination criterion for the bisection method can be given as an argument to the constructor (`None` signifies that no bisection algorithm is applied). Name of program file: `minmaxf2.py`.  $\diamond$

**Exercise 9.43.** *Make a calculus calculator class.*

Given a function  $f(x)$  defined on a domain  $[a, b]$ , the purpose of many mathematical exercises is to sketch the function curve  $y = f(x)$ , compute the derivative  $f'(x)$ , find local and global extreme points, and compute the integral  $\int_a^b f(x)dx$ . Make a class `CalculusCalculator` which can perform all these actions for any function  $f(x)$  using numerical differentiation and integration, and the method explained in Exercise 9.41 or 9.42 for finding extrema.

Here is an interactive session with the class where we analyze  $f(x) = x^2e^{-0.2x} \sin(2\pi x)$  on  $[0, 6]$  with a grid (set of  $x$  coordinates) of 700 points:

```
>>> from CalculusCalculator import *
>>> def f(x):
...     return x**2*exp(-0.2*x)*sin(2*pi*x)
...
>>> c = CalculusCalculator(f, 0, 6, resolution=700)
>>> c.plot()           # plot f
>>> c.plot_derivative() # plot f'
>>> c.extreme_points()

All minima: 0.8052, 1.7736, 2.7636, 3.7584, 4.7556, 5.754, 0
All maxima: 0.3624, 1.284, 2.2668, 3.2604, 4.2564, 5.2548, 6
Global minimum: 5.754
Global maximum: 5.2548

>>> c.integral
-1.7353776102348935
>>> c.df(2.51)      # c.df(x) is the derivative of f
-24.056988888465636
>>> c.set_differentiation_method(Central4)
>>> c.df(2.51)
-24.056988832723189
>>> c.set_integration_method(Simpson) # more accurate integration
>>> c.integral
-1.7353857856973565
```

Design the class such that the above session can be carried out.

Hint: Use classes from the `Diff` and `Integrator` hierarchies (Chapters 9.2 and 9.3) for numerical differentiation and integration (with, e.g., `Central2` and `Trapezoidal` as default methods for differentiation and integration, respectively). The method `set_differentiation_method` takes a subclass name in the `Diff` hierarchy as argument, and makes an attribute `df` that holds a subclass instance for computing derivatives. With `set_integration_method` we can similarly set the integration method as a subclass name in the `Integrator` hierarchy, and then compute the integral  $\int_a^b f(x)dx$  and store the value in the attribute `integral`. The `extreme_points` method performs a `print` on a `MinMax` instance, which is stored as an attribute in the calculator class. Name of program file: `CalculusCalculator.py`.

◇

**Exercise 9.44.** *Extend Exer. 9.43.*

Extend class `CalculusCalculator` from Exercise 9.43 to offer computations of inverse functions. A numerical way of computing inverse functions is explained in Chapter 5.1.10. Exercise 7.20 suggests an improved implementation using classes. Use the `InverseFunction` implementation from Exercise 7.20 in class `CalculusCalculator`. Name of program file: `CalculusCalculator2.py`.  $\diamond$

**Exercise 9.45.** *Formulate a 2nd-order ODE as a system.*

In this and subsequent exercises we shall deal with the following second-order ordinary differential equation with two initial conditions:

$$m\ddot{u} + f(\dot{u}) + s(u) = F(t), \quad t > 0, \quad u(0) = U_0, \quad \dot{u}(0) = V_0. \quad (9.54)$$

Write (9.54) as a system of two first-order differential equations. Also set up the initial condition for this system.

*Physical Applications.* Equation (9.54) has a wide range of applications throughout science and engineering. A primary application is damped spring systems in, e.g., cars and bicycles:  $u$  is the vertical displacement of the spring system attached to a wheel;  $\dot{u}$  is then the corresponding velocity;  $F(t)$  resembles a bumpy road;  $s(u)$  represents the force from the spring; and  $f(\dot{u})$  models the damping force (friction) in the spring system. For this particular application  $f$  and  $s$  will normally be linear functions of their arguments:  $f(\dot{u}) = \beta\dot{u}$  and  $s(u) = ku$ , where  $k$  is a spring constant and  $\beta$  some parameter describing viscous damping.

Equation (9.54) can also be used to describe the motions of a moored ship or oil platform in waves: the moorings act as a nonlinear spring  $s(u)$ ;  $F(t)$  represents environmental excitation from waves, wind, and current;  $f(\dot{u})$  models damping of the motion; and  $u$  is the one-dimensional displacement of the ship or platform.

Oscillations of a pendulum can be described by (9.54):  $u$  is the angle the pendulum makes with the vertical;  $s(u) = (mg/L)\sin(u)$ , where  $L$  is the length of the pendulum,  $m$  is the mass, and  $g$  is the acceleration of gravity;  $f(\dot{u}) = \beta|\dot{u}|\dot{u}$  models air resistance (with  $\beta$  being some suitable constant, see Exercises 1.10 and 9.50); and  $F(t)$  might be some motion of the top point of the pendulum.

Another application is electric circuits with  $u(t)$  as the charge,  $m = L$  as the inductance,  $f(\dot{u}) = R\dot{u}$  as the voltage drop accross a resistor  $R$ ,  $s(u) = u/C$  as the voltage drop accross a capacitor  $C$ , and  $F(t)$  as an electromotive force (supplied by a battery or generator).

Furthermore, Equation (9.54) can act as a simplified model of many other oscillating systems: aircraft wings, lasers, loudspeakers, microphones, tuning forks, guitar strings, ultrasound imaging, voice, tides, the El Niño phenomenon, climate changes – to mention some.

We remark that (9.54) is a possibly nonlinear generalization of Equation (C.8) explained in Appendix C.1.3. The case in Appendix C cor-

responds to the special choice of  $f(\dot{u})$  proportional to the velocity  $\dot{u}$ ,  $s(u)$  proportional to the displacement  $u$ , and  $F(t)$  as the acceleration  $\ddot{u}$  of the plate and the action of the gravity force.  $\diamond$

**Exercise 9.46.** *Solve the system in Exer. 9.45 in a special case.*

Make a function

```
def rhs(u, t):
    ...
```

for returning the right-hand side of the first-order differential equation system from Exercise 9.45. As usual, the `u` argument is an array or list with the two solution components `u[0]` and `u[1]` at some time `t`. Inside `rhs`, assume that you have access to three global Python functions `friction(dudt)`, `spring(u)`, and `external(t)` for evaluating  $f(\dot{u})$ ,  $s(u)$ , and  $F(t)$ , respectively.

Test the `rhs` function in combination with the functions  $f(\dot{u}) = 0$ ,  $F(t) = 0$ ,  $s(u) = u$ , and the choice  $m = 1$ . The differential equation then reads  $\ddot{u} + u = 0$ . With initial conditions  $u(0) = 1$  and  $\dot{u}(0) = 0$ , one can show that the solution is given by  $u(t) = \cos(t)$ . Apply two numerical methods: the 4th-order RungeKutta method and the Forward Euler method from the `ODESolver` module developed in Chapter 9.4. Use a time step  $\Delta t = \pi/20$ .

Plot  $u(t)$  and  $\dot{u}(t)$  versus  $t$  together with the exact solutions. Also make a plot of  $\dot{u}$  versus  $u$  (`plot(u[:,0], u[:,1])` if `u` is the array returned from the solver's `solve` method). In the latter case, the exact plot should be a circle<sup>17</sup>, but the ForwardEuler method results in a spiral. Investigate how the spiral develops as  $\Delta t$  is reduced.

The kinetic energy  $K$  of the motion is given by  $\frac{1}{2}m\dot{u}^2$ , and the potential energy  $P$  (stored in the spring) is given by the work done by the spring force:  $P = m \int_0^u s(v)dv = \frac{1}{2}mu^2$ . Make a plot with  $K$  and  $P$  as functions of time for both the 4th-order Runge-Kutta method and the Forward Euler method. In the present test case, the sum of the kinetic and potential energy should be constant. Compute this constant analytically and plot it together with the sum  $K + P$  as computed by the 4th-order Runge-Kutta method and the Forward Euler method.

Name of program file: `oscillator_v1.py`.  $\diamond$

**Exercise 9.47.** *Enhance the code from Exer. 9.46.*

The `rhs` function written in Exercise 9.46 requires that there is one particular set of Python functions `friction(dudt)`, `spring(u)`, and `external(t)` representing  $f(\dot{u})$ ,  $s(u)$ , and  $F(t)$ , respectively. One must also assume that a global variable `m` holds the value of  $m$ . Frequently, we want to work with different choices of  $f(\dot{u})$ ,  $s(u)$ , and  $F(t)$ , which with the `rhs` function proposed in Exercise 9.46 leads to `if` tests for

<sup>17</sup> The points on the curve are  $(\cos t, \sin t)$ , which all lie on a circle as  $t$  is varied.

the choices inside the `friction`, `spring`, and `external` functions. For example,

```
def spring(u):
    if spring_type == 'linear':
        return k*u
    elif spring_type == 'cubic':
        return k*(u - 1./6*u**3)
```

It would be better to introduce two different `spring` functions instead, or represent these functions by classes as explained in Chapter 7.1.2.

Instead of the `rhs` function in Exercise 9.46, develop a class `RHS` where the constructor takes the  $f(\dot{u})$ ,  $s(u)$ , and  $F(t)$  functions as arguments `friction`, `spring`, and `external`. The  $m$  parameter must also be an argument. Use a `__call__` method to evaluate the right-hand side of the differential equation system arising from (9.54).

Write a function

```
def solve(T,
         dt,
         initial_u,
         initial_dudt,
         method=RungeKutta4,
         m=1.0,
         friction=lambda dudt: 0,
         spring=lambda u: u,
         external=lambda t: 0):
    ...
    return u, t
```

for solving (9.54) from time zero to some stopping time  $T$  with time step  $dt$ . The other arguments hold the initial conditions for  $u$  and  $\dot{u}$ , the class for the numerical solution method, as well as the  $f(\dot{u})$ ,  $s(u)$ , and  $F(t)$  functions. (Note the use of lambda functions, see Chapter 2.2.11, to quickly define some default choices for  $f(\dot{u})$ ,  $s(u)$ , and  $F(t)$ ). The `solve` function must create an `RHS` instance and feed this to an instance of the class referred to by `method`.

Also write a function

```
def makeplot(T,
            dt,
            initial_u,
            initial_dudt,
            method=RungeKutta4,
            m=1.0,
            friction=lambda dudt: 0,
            spring=lambda u: u,
            external=lambda t: 0,
            u_exact=None):
```

which calls `solve` and makes plots of  $u$  versus  $t$ ,  $\dot{u}$  versus  $t$ , and  $\dot{u}$  versus  $u$ . If `u_exact` is not `None`, this argument holds the exact  $u(t)$  solution, which should then be included in the plot of the numerically computed solution.

Make a function `get_input`, which reads input data from the command line and calls `makeplot` with these data. Use option-value pairs on the command line to specify `T`, `dt`, `initial_u`, `initial_dudt`, `m`, `method`, `friction`, `spring`, `external`, and `u_exact`. Use `eval` on the first five values so that mathematical expressions like `pi/10` can be specified. Also use `eval` on `method` to transform the string with a class name into a Python class object. For the latter four arguments, assume the command-line value is a string that can be turned into a function via the `StringFunction` tool from Chapter 3.1.4. Let string formulas for `friction`, `spring`, and `external` have `dudt`, `u`, and `t` as independent variables, respectively. For example,

```
elif option == '--friction':
    friction = StringFunction(value, independent_variable='dudt')
```

The `friction`, `spring`, and `external` functions will be called with a scalar (real number) argument, while it is natural to call `u_exact` with an array of time values. In the latter case, the `StringFunction` object must be vectorized (see Chapter 4.4.3):

```
elif option == '--u_exact':
    u_exact = StringFunction(value, independent_variable='t')
    u_exact.vectorize(globals())
```

Collect the functions in a module, and let the test block in this module call the `get_input` function. Test the module by running the tasks from Exercise 9.46:

---

Terminal

---

```
oscillator_v2.py --method ForwardEuler --u_exact "cos(t)" \
--dt "pi/20" --T "5*pi"
oscillator_v2.py --method RungeKutta4 --u_exact "cos(t)" \
--dt "pi/20" --T "5*pi"
oscillator_v2.py --method ForwardEuler --u_exact "cos(t)" \
--dt "pi/40" --T "5*pi"
oscillator_v2.py --method ForwardEuler --u_exact "cos(t)" \
--dt "pi/80" --T "5*pi"
```

---

A demo with friction and external forcing can also be made, for example,

---

Terminal

---

```
oscillator_v2.py --method RungeKutta4 --friction "0.1*dudt" \
--external "sin(0.5*t)" --dt "pi/80" --T "40*pi" --m 10
```

---

Name of program file: `oscillator_v2.py`. ◇

**Exercise 9.48.** *Make a tool for analyzing oscillatory solutions.*

The solution  $u(t)$  of the equation (9.54) often exhibit an oscillatory behaviour (for the test problem in Exercise 9.46 we have that  $u(t) = \cos t$ ). It is then of interest to find the wavelength of the oscillations. The purpose of this exercise is to find and visualize the distance between peaks in a numerical representation of a continuous function.

Given an array  $(y_0, \dots, y_{n-1})$  representing a function  $y(t)$  sampled at various points  $t_0, \dots, t_{n-1}$ . A local maximum of  $y(t)$  occurs at  $t = t_k$  if  $y_{k-1} < y_k > y_{k+1}$ . Similarly, a local minimum of  $y(t)$  occurs at  $t = t_k$  if  $y_{k-1} > y_k < y_{k+1}$ . By iterating over the  $y_1, \dots, y_{n-2}$  values and making the two tests, one can collect local maxima and minima as  $(t_k, y_k)$  pairs. Make a function `minmax(t, y)` which returns two lists, `minima` and `maxima`, where each list holds pairs (2-tuples) of  $t$  and  $y$  values of local minima or maxima. Ensure that the  $t$  value increases from one pair to the next. The arguments `t` and `y` in `minmax` hold the coordinates  $t_0, \dots, t_{n-1}$  and  $y_0, \dots, y_{n-1}$ , respectively.

Make another function `wavelength(peaks)` which takes a list `peaks` of 2-tuples with  $t$  and  $y$  values for local minima or maxima as argument and returns an array of distances between consecutive  $t$  values, i.e., the distances between the peaks. These distances reflect the local wavelength of the computed  $y$  function. More precisely, the first element in the returned array is `peaks[1][0]-peaks[0][0]`, the next element is `peaks[2][0]-peaks[1][0]`, and so forth.

Test the `minmax` and `wavelength` functions on  $y$  values generated by  $y = e^{t/4} \cos(2t)$  and  $y = e^{-t/4} \cos(t^2/5)$  for  $t \in [0, 4\pi]$ . Plot the  $y(t)$  curve in each case, and mark the local minima and maxima computed by `minmax` with circles and boxes, respectively. Make a separate plot with the array returned from the `wavelength` function (just plot the array against its indices - the point is to see if the wavelength varies or not). Plot only the wavelengths corresponding to maxima.

Make a module with the `minmax` and `wavelength` function, and let the test block perform the tests specified above. Name of program file: `wavelength.py`.  $\diamond$

#### Exercise 9.49. Replace functions by class in Exer. 9.46.

The three functions `solve`, `makeplot`, and `get_input` from Exercise 9.46 contain a lot of arguments. Instead of shuffling long argument lists into functions, we can create classes that hold the arguments as attributes.

Introduce three classes: `Problem`, `Solver`, and `Visualize`. In class `Problem`, we store the specific data about the problem to be solved, in this case the parameters `initial_u`, `initial_dudt`, `m`, `friction`, `spring`, `external`, and `u_exact`, using the namings in Exercise 9.46. Methods can read the user's values from the command line and initialize attributes, and form the right-hand side of the differential equation system to be solved.

In class `Solver`, we store the data related to solving a system of ordinary differential equations: `T`, `dt`, and `method`, plus the solution. Methods can read input from the command line and initialize attributes, and solve the ODE system using information from a class `Problem` instance.



The final class, `Visualize`, has attributes holding the solution of the problem and can make various plots. We may control the type of plots by command-line arguments.

Class `Problem` may look like

```
class Problem:
    def initialize(self):
        """Read option-value pairs from sys.argv."""
        self.m = eval(read_cml('--m', 1.0))
        ...
        s = read_cml('--spring', '0')
        self.spring = StringFunction(s, independent_variable='u')
        ...
        s = read_cml('--u_exact', '0')
        if s != '0':
            self.u_exact = None
        else:
            self.u_exact = \
                StringFunction(s, independent_variable='t')
            self.u_exact.vectorize(globals())
        ...

    def rhs(self, u, t):
        """Define the right-hand side in the ODE system."""
        m, f, s, F = \
            self.m, self.friction, self.spring, self.external
        u, dudt = u
        return [dudt,
                (1./m)*(F(t) - f(dudt) - s(u))]
```

The `initialize` method calls `read_cml` from `scitools.misc` to extract the value proceeding the option `-m`. We could have used the `getopt` module, but we aim at reading data from the command line in separate phases in the various classes, and `getopt` does not allow reading the command line more than once. Therefore, we have to use a specialized function `read_cml`.

The exemplified call to `read_cml` implies to look for the command-line argument `-m` for  $m$  and treat the next command-line argument as the value of  $m$ . If the option `-m` is not found at the command line, we use the second argument in the call (here `self.m`) as default value, but returned as a string (since values at the command line are strings).

The class `Solver` follows the design of class `Problem`, but it also has a `solve` method that solves the problem and stores the solution  $u$  of the ODEs and the time points  $t$  as attributes:

```
class Solver:
    def initialize(self):
        self.T = eval(read_cml('--T', 4*pi))
        self.dt = eval(read_cml('--dt', pi/20))
        self.method = eval(read_cml('--method', 'RungeKutta4'))

    def solve(self, problem):
        self.solver = self.method(problem.rhs, self.dt)
        ic = [problem.initial_u, problem.initial_dudt]
        self.solver.set_initial_condition(ic, 0.0)
        self.u, self.t = self.solver.solve(self.T)
```

The use of `eval` to initialize `self.T` and `self.dt` allows us to specify these parameters by arithmetic expressions like `pi/40`. Using `eval` on the string specifying the numerical method turns this string into a class type (i.e., a name 'ForwardEuler' is turned into the class `ForwardEuler` and stored in `self.method`).

The `Visualizer` class holds references to a `Problem` and `Solver` instance and creates plots. The user can specify plots in an interactive dialog in the terminal window. Inside a loop, the user is repeatedly asked to specify a plot until the user responds with `quit`. The specification of a plot can be one of the words `u`, `dudt`, `dudt-u`, `K`, and `wavelength` which means a plot of  $u(t)$  versus  $t$ ,  $\dot{u}(t)$  versus  $t$ ,  $\dot{u}$  versus  $u$ ,  $K$  versus  $t$ , and  $u$ 's wavelength versus its indices, respectively. The wavelength can be computed from the local maxima of  $u$  as explained in Exercise 9.48).

A sketch of class `Visualizer` is given next:

```
class Visualizer:
    def __init__(self, problem, solver):
        self.problem = problem
        self.solver = solver

    def visualize(self):
        t = self.solver.t    # short form
        u, dudt = self.solver.u[:,0], self.solver.u[:,1]

        # tag all plots with numerical and physical input values:
        title = 'solver=%s, dt=%g, m=%g' % \
            (self.solver.method, self.solver.dt, self.problem.m)
        # can easily get the formula for friction, spring and force
        # if these are string formulas:
        if isinstance(self.problem.friction, StringFunction):
            title += ' f=%s' % str(self.problem.friction)
        if isinstance(self.problem.spring, StringFunction):
            title += ' s=%s' % str(self.problem.spring)
        if isinstance(self.problem.external, StringFunction):
            title += ' F=%s' % str(self.problem.external)

        plot_type = ''
        while plot_type != 'quit':
            plot_type = raw_input('Specify a plot: ')
            figure()
            if plot_type == 'u':
                # plot u vs t
                if self.problem.u_exact is not None:
                    hold('on')
                # plot self.problem.u_exact vs t
                show()
                hardcopy('tmp_u.eps')
            elif plot_type == 'dudt':
                ...
```

Make a complete implementation of the three proposed classes. Also make a `main` function that (i) creates a problem, solver, and visualizer, (ii) initializes the problem and solver with input data from the command line, (iii) calls the solver, and (iv) calls the visualizer's `visualize` method to create plots. Collect the classes and functions in a module

`oscillator`, which has a call to `main` in the test block. The first task from Exercises 9.46 or 9.47 can now be run as

---

Terminal

---

```
oscillator.py --method ForwardEuler --u_exact "cos(t)" \
--dt "pi/20" --T "5*pi"
```

---

The other tasks from Exercises 9.46 or 9.47 can be tested similarly.

Explore some of the possibilities of specifying several functions on the command line:

---

Terminal

---

```
oscillator.py --method RungeKutta4 --friction "0.1*dudt" \
--external "sin(0.5*t)" --dt "pi/80" \
--T "40*pi" --m 10

oscillator.py --method RungeKutta4 --friction "0.8*dudt" \
--external "sin(0.5*t)" --dt "pi/80" \
--T "120*pi" --m 50
```

---

We remark that this module has the same physical and numerical functionality as the module in Exercise 9.47. The only difference is that the code in the modules is organized differently. The organization in terms of classes in the present module avoids shuffling lots of arguments to functions and is often viewed as superior. When solving more complicated problems that result in much larger codes, the class version is usually much simpler to maintain and extend. The reason is that variables are packed together in a few units along with the functionality that operates on the variables.

We also remark that it can be difficult to get `pyreport` to work properly with the present module and further use of it in the forthcoming exercises.

Name of program file: `oscillator.py`. ◇

**Exercise 9.50.** *Allow flexible choice of functions in Exer. 9.49.*

Some typical choices of  $f(\dot{u})$ ,  $s(u)$ , and  $F(t)$  in (9.54) are listed below:

1. Linear friction force (low velocities):  $f(\dot{u}) = 6\pi\mu R\dot{u}$  (Stokes drag), where  $R$  is the radius of a spherical approximation to the body's geometry, and  $\mu$  is the viscosity of the surrounding fluid.
2. Quadratic friction force (high velocities):  $f(\dot{u}) = \frac{1}{2}C_D\rho A|\dot{u}|\dot{u}$ , see Exercise 1.10 for explanation of symbols.
3. Linear spring force:  $s(u) = ku$ , where  $k$  is a spring constant.
4. Sinusoidal spring force:  $s(u) = k\sin u$ , where  $k$  is a constant.
5. Cubic spring force:  $s(u) = k(u - \frac{1}{6}u^3)$ , where  $k$  is a spring constant.
6. Sinusoidal external force:  $F(t) = F_0 + A\sin\omega t$ , where  $F_0$  is the mean value of the force,  $A$  is the amplitude, and  $\omega$  is the frequency.
7. "Bump" force:  $F(t) = H(t - t_1)(1 - H(t - t_2))F_0$ , where  $H(t)$  is the Heaviside function from Exercise 2.36,  $t_1$  and  $t_2$  are two given time

- points, and  $F_0$  is the size of the force. This  $F(t)$  is zero for  $t < t_1$  and  $t > t_2$ , and  $F_0$  for  $t \in [t_1, t_2]$ .
8. Random force 1:  $F(t) = F_0 + A \cdot U(t; B)$ , where  $F_0$  and  $A$  are constants, and  $U(t; B)$  denotes a function whose value at time  $t$  is random and uniformly distributed in the interval  $[-B, B]$ .
  9. Random force 2:  $F(t) = F_0 + A \cdot N(t; \mu, \sigma)$ , where  $F_0$  and  $A$  are constants, and  $N(t; \mu, \sigma)$  denotes a function whose value at time  $t$  is random, Gaussian distributed number with mean  $\mu$  and standard deviation  $\sigma$ .

Make a module `functions` where each of the choices above are implemented as a class with a `__call__` special method. Also add a class `Zero` for a function whose value is always zero. It is natural that the parameters in a function are set as arguments to the constructor. The different classes for spring functions can all have a common base class holding the  $k$  parameter as attribute. Name of program file: `functions.py`.  $\diamond$

**Exercise 9.51.** Use the modules from Exer. 9.49 and 9.50.

The purpose of this exercise is to demonstrate the use of the classes from Exercise 9.50 to solve problems described by (9.54).

With a lot of models for  $f(\dot{u})$ ,  $s(u)$ , and  $F(t)$  available as classes in `functions.py`, it is more challenging to read information about these mathematical functions from the command line<sup>18</sup>. We therefore propose to create the relevant instances in the program and assign them directly to attributes in the `Problem` instance, e.g.,

```
problem = Problem()
problem.m = 1.0
k = 1.2
problem.spring = CubicSpring(k)
...
```

The solver and visualizer objects can still be initialized from the command line, if desired.

Make a separate file, say `oscillator_test.py`, where you import class `Problem`, `Solver`, and `Visualizer`, plus all classes from the `functions` module. Provide a `main1` function with initializations of class `Problem` attributes as indicated above for the case with a Forward Euler method,  $m = 1$ ,  $u(0) = 1$ ,  $\dot{u}(0) = 0$ , no friction (use class `Zero`), no external forcing (class `Zero`), a linear spring  $s(u) = u$ ,  $\Delta t = \pi/20$ ,  $T = 8\pi$ , and exact  $u(t) = \cos(t)$ .

Then make another function `main2` for the case with a 4th-order Runge-Kutta method,  $m = 5$ ,  $u(0) = 1$ ,  $\dot{u}(0) = 0$ , linear friction  $f(\dot{u}) = 0.1\dot{u}$ ,  $s(u) = u$ ,  $F(t) = \sin(\frac{1}{2}t)$ ,  $\Delta t = \pi/80$ ,  $T = 60\pi$ , and no knowledge of an exact solution. Let the test block use the first command-line argument to indicate a call to `main1` or `main2`. Name of program file: `oscillator_test.py`.  $\diamond$

<sup>18</sup> It can be easily done, however, using `read_cml_func` from `scitools.misc`.

**Exercise 9.52.** *Use the modules from Exer. 9.49 and 9.50.*

Make a program `oscillator_conv.py` where you import the `Problem` and `Solver` classes from the `oscillator` module in Exercise 9.50 and implement a loop in which  $\Delta t$  is reduced. The end result should be a plot with the curves  $u$  versus  $t$  corresponding to the various  $\Delta t$  values. Typically, we want to do something like

```
from oscillator import Problem, Solver
from scitools.std import plot, hold, hardcopy

problem = Problem()
problem.initialize()
solver = Solver()
solver.initialize()

# see how the solution changes by halving dt n times:
n = 4
for k in range(n):
    solver.solve(problem)
    u, t = solver.u[:,0], solver.t
    # plot u
    solver.dt = solver.dt/2.0
```

Extend this program with another loop over increasing  $m$  values.

Hopefully, you will realize how flexible the classes from Exercises 9.49 and 9.50 are for solving a variety of problems. We can give a set of physical and numerical parameters in a flexible way on the command line, and in the program we may make loops or other constructions to manipulate the input data further.

Name of program file: `oscillator_conv.py`.

◇