Chapter 6
About Scientific Software

When a science problem is solved with the aid of numerical computations, the solution procedure involves several steps:

1. Understanding the problem and formulating a mathematical model
2. Using numerical methods to solve the mathematical problems
3. Implementing the numerical methods in a computer program
4. Verifying that the results from the program are mathematically correct
5. Applying the program to the scientific problem and interpreting the results

Normally, this is a repetitive cycle: interpretation of the results often leads to adjustments in the mathematical model, numerical methods, and the computer program.

The cycle listed above typically has a “theory” part and a “practice” part. Most books emphasize theory, i.e., deriving and analyzing mathematical models and numerical methods. The practice part, consisting of translating the models and methods to running code, producing numbers, and verifying the results, is equally important and requires skills that must be developed systematically. The present chapter provides the first steps toward gaining the necessary skills.

Scientific Software Requirements

Software performing scientific computations must be

1. Mathematically correct
2. Efficient (speed, memory usage)
3. Easy to maintain and extend

If there is an error in the program, the calculations will most likely be wrong and the results will become useless. Many types of numerical computations demand days or weeks of computing time and the combined memory of a large collection of computers. Efficiency with respect to speed and memory usage is thus of utmost importance. Unfortunately, many efficiency improvements also easily introduce errors in the code. The complexity of scientific software has reached the limit
where maintenance and future extensions have become very difficult, and the modification of complicated codes easily leads to a significant danger of introducing errors. Careful design of scientific software systems is therefore necessary.

Software Development Skills

The scientific software developer needs several skills to meet the demands of the previous paragraph. These skills include

1. Understanding the mathematical problem to be solved
2. Understanding the numerical methods to be used
3. Designing appropriate algorithms and data structures
4. Selecting the most suitable programming language and tools
5. Using libraries
6. Verifying the correctness of the results

The first two points are critical to the last point. Ideally, the software developer should also have an understanding of the physical problem being solved, but as long as the mathematical model for the physical problem is specified completely, the software development (in terms of programming and verification) is decoupled from the original problem. In fact, such a decoupling encourages the production of software that can be applied to a range of different physical problem areas. Points 3–5 are closely tied and have often been ignored in the literature. One reason may be that up until the 1990s almost all scientific software developers used Fortran as the programming tool. The tendency now is to use a collection of tools to solve a given problem, i.e., the developer needs to select the right tool for each subtask. This requires a knowledge of a range of tools.

Scientific software development, and especially the testing phase, is known to be very time consuming and the number one reason why budgets are so frequently exceeded in scientific computing projects. Students also tend to spend much time on going from the mathematics to a working code. The rule of thumb is therefore to avoid developing numerical software if possible, i.e., one should reuse existing software to as large extent as possible. For some problem areas, there are software packages providing all functionality you need in the solution process, and there is no need for software development. Most scientific problems, however, demand some kind of programming, fortunately in the form of calling up available functionality in various libraries. How to do this efficiently again requires knowledge of different programming tools.

This Chapter

Fortunately, there are many techniques to speed up the development of numerical software and increase the reliability of an implementation. The present chapter gives
a first introduction to writing scientific software, with emphasis on two aspects:

- The reliable translation of mathematical algorithms into working code
- A glimpse of some relevant programming languages and tools

We work with very simple numerical sample problems such that the type of problem and its solution methods are hopefully well understood. This enables us to concentrate on software issues. However, many of the important software issues listed above are beyond the scope of this chapter, because they do not become evident before one tries to implement sets of complicated algorithms.

### 6.1 Algorithms Expressed as Pseudo Code

Let us first address the issue of creating a program without errors. We suggest approaching the challenge in two steps:

1. Express the numerical problem to be solved and the methods to be used in a complete algorithm.
2. Translate the algorithm into a computer code using a specific programming language.

Jumping directly from a compact description of the numerical problem to computer programming is often an error-prone procedure, even if you are very experienced with both the numerics of the problem and software development. The smaller you can make the gap between an algorithm, expressed in mathematical terms on a piece of paper, and a computer program, the easier it will be to develop the program and check it for correctness. A direct consequence of this strategy is that we end most of the discussion of numerical methods in this book with detailed, complete algorithms. In most cases it will be straightforward to translate the algorithm into a program, provided you are experienced with programming, which is taken as a prerequisite when working with this book.

We emphasize the adjective complete in our phrase complete algorithms; this means that all numerical details are covered in the algorithm such that the implementation only involves a translation and no numerical or mathematical issues.

The art of formulating algorithms sufficiently close to computer programs is best illustrated through examples. Two simple problem areas are considered here: numerical integration by the trapezoidal rule and solution of an ordinary differential equation (ODE) by the Heun scheme.

#### 6.1.1 Basic Ingredients of Pseudo Codes

The mapping of a physical problem into computer code has some characteristics that are independent of the problem being solved. First, the physical problem must be expressed as a mathematical model, i.e., a set of mathematical problems to be solved. For each mathematical problem we need to select an appropriate numerical
solution method. Each method is expressed as an algorithm and implemented as a piece of code, normally a function. The code is then a collection of functions implementing the various algorithms (solution steps) of the whole problem. The communication between the functions depends on what type of data structures we use. Therefore, creating numerical software is an interplay between algorithms and data structures. Typical data structures are scalar variables, arrays, and functions.

A complete algorithm is normally expressed as pseudo code, which is a mixture of mathematical formulas and instructions known from computer languages. The purpose is to get an algorithm that easily translates into a computer program. Some important instructions used in pseudo code (and computer programs) are

1. Assignments; the notation \( s \leftarrow s + 2 \) means that we assign the value of the expression \( s + 2 \) to \( s \), i.e., \( s \) is over-written by a new value,
2. For loops\(^2\); loops controlled by a counter running between two values with a certain step length,
3. While loops; loops controlled by a boolean condition,
4. Functions\(^3\); subprograms taking a set of variables as input (arguments) and returning a set of variables,
5. Arrays; sequences of numbers, such as \( u_1, u_2, \ldots, u_{10} \).

As an example, consider the sum

\[
\sum_{i=1}^{n-1} f(a + ih),
\]

where \( f \) is a function of a scalar variable, and \( a \) and \( h \) are constants. We want to express this sum in a way that translates more or less directly to statements in a computer program. Normally, a sum is computed by a for loop, using a variable (here \( s \)) that accumulates the individual contributions \( f(a + ih) \) to the sum:

\[
s = 0
\]
\[
\text{for } i = 1, \ldots, n - 1
\]
\[
s \leftarrow s + f(a + ih)
\]
\[
\text{end for}
\]

Note that the parameters \( a \) and \( h \) must be pre-computed for this algorithm to work. After the loop, \( s \) equals \( \sum_{i=1}^{n-1} f(a + ih) \).

Instead of a for loop, we can use a while loop to implement the sum:

\[
s = 0
\]
\[
i = 1
\]

\[
\text{while } i < n
\]
\[
s \leftarrow s + f(a + ih)
\]
\[
i = i + 1
\]
\[
\text{end while}
\]

\(^1\)Most programming languages would just use a notation such as \( s=s+2 \), but this notation is mathematically incorrect, so we prefer to use the left arrow \( \leftarrow \) in our mathematically-oriented algorithmic notation.

\(^2\)Also called do loops (named after the corresponding Fortran construction).

\(^3\)Also called subprogram, subroutine, or procedure.
The forthcoming sections provide more examples.

### 6.1.2 Integration

Suppose we want to integrate a function \( f(x) \) from \( x = a \) to \( x = b \) with the aid of the composite trapezoidal RULE (derived in Sect. 1.3). Mathematically, we can express this method for computing an integral as follows:

\[
\int_a^b f(x) \, dx \approx \frac{h}{2} f(a) + \frac{h}{2} f(b) + \sum_{i=1}^{n-1} hf(a + ih), \quad h = \frac{b - a}{n}. \tag{6.1}
\]

This is a compact formulation that contains all the necessary information for calculating the integral on a computer. There are some steps from (6.1) to running code, but you have hopefully already tried to implement methods such as (6.1) while working with exercises in the introductory chapter. Now it is time to revise such implementational work and adopt good habits.

Before thinking of an implementation, we should express a compact formula such as (6.1) as an algorithm, i.e., as a set of steps that naturally translate into similar steps in a computer code. The complete algorithm corresponding to the mathematical formulation (6.1) can be expressed as follows:

\[
\begin{align*}
& h = \frac{b - a}{n} \\
& s = 0 \\
& \text{for } i = 1, \ldots, n - 1 \\
& \quad s \leftarrow s + hf(a + ih) \\
& \text{end for} \\
& s \leftarrow s + \frac{h}{2} f(a) + \frac{h}{2} f(b)
\end{align*}
\]

The nature of this algorithm is that we provide \( a, b, n, \) and \( f(x) \) as input and get the answer \( s \) as output. Such input–output algorithms are often conveniently expressed in a way that resembles the concept of functions or subprograms in programming languages.

For example, we could write the algorithms in a notation such as

\[
\text{trapezoidal } (a, b, f, n) \\
\text{... do something ... store final result in } s \\
\text{return } s
\]

This notation is hopefully self-explanatory for anyone having some experience with computer programming; we explicitly indicate that \( a, b, f, \) and \( n \) are input data and \( s \) the output, i.e., the result of the algorithm.
The complete algorithm for the trapezoidal rule, using the proposed notation, appears separately as Algorithm 6.1.

**Algorithm 6.1**

**Trapezoidal Integration.**

\[
\begin{align*}
\text{trapezoidal}(a, b, f, n) \\
h &= \frac{b-a}{n} \\
s &= 0 \\
\text{for } i = 1, \ldots, n - 1 \\
&\quad s \leftarrow s + hf(a + ih) \\
\text{end for} \\
&\quad s \leftarrow s + \frac{h}{2} f(a) + \frac{h}{2} f(b) \\
\text{return } s 
\end{align*}
\]

The notation used in Algorithm 6.1 is a typical example of mathematical pseudo code.

Having the algorithm available as mathematical pseudo code makes the step toward a working computer code quite small. Some readers might think that the step is so small that we could equally well have written the lines directly in a specific computer language. This argument is relevant in the present example, where there is hardly any complexity of the algorithm implied by (6.1). In more complicated problems, however, the mathematical pseudo code is still quite close to the mathematical exposition of the numerical methods, whereas a computer code will contain many more details specific to the chosen language. Experience shows that the two-step procedure of first deriving a correct mathematical pseudo code and then using it as a reference when translating, and later checking, the computer implementation simplifies software development considerably.

Another attractive feature of the mathematical pseudo code is that we turn numerical solution procedures into a form that is ready for implementation. However, the choice of programming language, programming style, data types, and computing instructions (which differ highly between individuals and organizations) remains open.

### 6.1.3 Optimization of Algorithms and Implementations

There is a strong tradition in scientific computing of carefully examining algorithms with the aim of reducing the number of arithmetic operations and the need for storing computed information. This is because advanced problems in scientific computing require optimization of speed and memory usage.
Saving Arithmetic Operations

Examining the algorithm above, we see that the factor \( h \) appears in all terms. This is perhaps even more obvious from the mathematical formula (6.1). We can avoid \( n \) multiplications by \( h \) through a factorization of (6.1). We can also be more careful with the factor \( 1/2 \); division is often slower than multiplication (perhaps by a factor of 4, depending on the hardware). A computationally more efficient version of (6.1) therefore reads

\[
\int_a^b f(x) dx \approx h \left\{ 0.5(f(a) + f(b)) + \sum_{i=1}^{n-1} f(a + ih) \right\}, \quad h = \frac{b-a}{n}.
\]  

(6.2)

To really see why (6.2) is more efficient than (6.1) we can count the number of operations: additions, subtractions, multiplications, divisions, and \( f(x) \) function calls. We find that (6.1) has \( 2n \) multiplications, \( 2n \) additions, three divisions, and \( n + 1 \) function calls. The formula in (6.2) implies \( n + 1 \) multiplications, \( 2n \) additions, one division, and \( n + 1 \) function calls. If \( f(x) \) is a complicated function requiring many arithmetic operations, such as \( \sin x \) or \( e^x \), evaluation of \( f \) will dominate the work of the algorithm. In that case there are minor differences between (6.1) and (6.2); the total work is well approximated by \( n + 1 \) calls to the function \( f \).

It is common to introduce floating-point operation (FLOP) as a synonym for addition, subtraction, multiplication, or division. However, this can be misleading on some hardware where division is significantly slower than the three other operations. Counting one multiplication/division and one addition/subtraction as a single floating-point operation is also common, because modern CPUs often have the possibility of running pieces of a compound expression in parallel.

We can avoid \( n - 1 \) multiplications \( ih \) in (6.2) by incrementing the function argument by \( h \) in each pass in the for loop used to compute the sum. The mathematical pseudo code incorporating this trick in (6.2) is expressed in Algorithm 6.2. The total work is now one multiplication, \( 2n \) additions, one division, and \( n + 1 \) function calls.

**Algorithm 6.2**

Optimized Trapezoidal Integration.

\[
\text{trapezoidal} \ (a, \ b, \ f, \ n)
\]

\[
h = \frac{b-a}{n}
\]

\[
s = 0
\]

\[
x = a
\]

for \( i = 1, \ldots, n - 1 \)

\[
x \leftarrow x + h
\]

\[
s \leftarrow s + f(x)
\]

end for

\[
s \leftarrow s + 0.5(f(a) + f(b))
\]

\[
s \leftarrow hs
\]

return \( s \)
When performing a single numerical integration of a function $f(x)$ with a “reasonable” $n$, the computations will, on today’s machines, be so fast and use so little memory that this latter optimization has no practical advantage; you get the answer immediately anyway. In fact, rewriting from (6.1) to (6.2) and then to Algorithm 6.2 just increases the probability of introducing errors. Nevertheless, if this integration is used inside a huge scientific computing code and is called billions of times, even a small speed improvement may be significant.

Example 6.1. The payoff of the optimization depends heavily on the function $f(x)$. If $f(x)$ is complicated, say, an evaluation of $f$ requires 100 multiplications (which is relevant when $f$ contains fundamental functions such as sin and log), saving a multiplication by $h$ can only improve the speed by 1%. To be specific, we tried to integrate

$$f_1(x) = e^{-x^2} \log(1 + x \sin x)$$

and

$$f_2(x) = 1 + x$$

from $a = 0$ to $b = 2$ with $n = 1,000$, repeated 10,000 times. In Fortran 77 implementations of Algorithms 6.1 and 6.2 there were no differences between the two versions of the algorithms when integrating $f_1(x)$. Even with $f_2(x)$, Algorithm 6.2 was only slightly faster than Algorithms 6.1. On the other hand, the total execution time increased by a factor of 10 when we switched from the simple $f_2$ function to the more complicated $f_1$ function.

Rely on Compiler Optimization Technology

Why was the benefit of our hand optimizations so small in the previous example? Administering the loop $i = 1, \ldots, n - 1$ and calling a function are expensive operations in computer programs, so saving a couple of multiplications inside a loop can drown in other tasks. Nevertheless, a much more important explanation why the benefit was so small has to do with compiler optimization technology. We turned on the maximum optimization of the GNU Fortran 77 compiler when compiling the codes corresponding to Algorithms 6.1 and 6.2. The Fortran compiler has 50 years of experience with optimizing loops involving mathematical expressions. Loops of the very simple type encountered in the present example can probably be analyzed to the fullest extent by most compilers. This means that the compiler will detect when we perform unnecessary multiplications by $h$. The compiler will probably also see that the function evaluation $f(a + ih)$ can be optimized, as we did by hand. These assertions are supported by compiling the codes without optimization; the CPU times of Algorithms 6.1 and 6.2 differed by a factor of almost 2, when integrating $f_2$. Negligible differences arose when integrating $f_1$, since the function evaluations dominate in this case.

---

4 See page 221 for how to measure CPU time.
Based on this simple optimization example, and our experience with scientific software development in general, we can provide some guidelines on how to implement numerical methods:

1. Use an algorithm that is easy to understand and that can easily be looked up in the literature.
2. Create a computer code that is as close to the algorithm as possible, such that a “one-to-one correspondence” can be checked by reading the code and the algorithm side by side.
3. Test the implementation on a simple problem where the exact answer is known (such as $f_2(x) = 1 + x$ in the previous example).
4. Be careful with hand optimizations before the code is verified.
5. As soon as some hand optimizations are implemented, compile the code with compiler optimization turned on, run the original and the hand-optimized code in a relevant application, check that the results are equivalent, and then compare timings.

Our recommended rule of thumb is to avoid optimization in the early stages of software development since non-optimized code is usually easier to understand and hence easier to debug. If CPU time consumption of the implemented algorithm is a great concern, one can proceed with rewriting the algorithm for optimization. Having the safe, non-optimized version at hand, it is much easier to verify the usually trickier optimized version. We agree with the famous computer scientist Donald Knuth, who said that “premature optimization is the root of all evil”.

We should emphasize that there are numerous examples, the first one appearing already in Sect. 6.1.4, especially in advanced scientific computing problems, where a human is better than a compiler to perform optimizations. Most often, however, the human’s contribution to faster code is to choose a more efficient numerical method, whereas the benefit of the compiler is to arrange the sequence of arithmetic operations in an optimal manner.

### 6.1.4 Developing Algorithms for Simpson’s Rule

Let us discuss another example regarding the derivation and implementation of algorithms for numerical integration. Simpson’s rule for numerical integration reads (see Project 1.7.2)

$$
\int_{a}^{b} f(x) \, dx \approx h \sum_{i=1}^{n} \left\{ \frac{1}{6} f(x_{i-1}) + \frac{4}{6} f(x_{i-\frac{1}{2}}) + \frac{1}{6} f(x_{i}) \right\},
$$

(6.3)

where

$$
x_{i} = a + ih, \quad x_{i-\frac{1}{2}} = \frac{1}{2}(x_{i-1} + x_{i}), \quad h = \frac{b-a}{n}.
$$

Simpson’s rule is more accurate than the trapezoidal rule, in the sense that it requires less computer power to calculate an integral with a certain accuracy. The task now is to implement Simpson’s rule.
Mathematical Pseudo Code

As we have learned, the mathematical description of the method in (6.3) should first be turned into a mathematical pseudo algorithm. Then we can proceed with translating the algorithm into computer language. With our recent experience in writing algorithms for the trapezoidal rule, it should be quite easy to come up with Algorithm 6.3. Note that we have not used the notation \(x_{i-1}, x_{i-\frac{1}{2}},\) and \(x_i,\) but \(x^-\), \(x\), and \(x^+\) instead. This is due to the fact that mathematical symbols with indices, such as \(x_i\), are often translated to arrays in computer codes. In the present case, we do not need to store the \(x_i\) values in an array, we just need to compute the \(x\) value inside the loop and then call \(f(x)\). Therefore, to avoid indices, we work with \(x^-, x,\) and \(x^+\). In a computer code these variables could be given names such as \(xm, x,\) and \(xp\).

Algorithm 6.3

\[
\text{Simpson's Rule.}
\]

\[
\begin{align*}
\text{Simpson } (a, b, f, n) \\
\quad & h = \frac{b-a}{n} \\
\quad & s = 0 \\
\quad & \text{for } i = 1, \ldots, n \\
\quad & \quad x^- = a + (i-1)h \\
\quad & \quad x^+ = a + ih \\
\quad & \quad x = \frac{1}{2}(x^- + x^+) \\
\quad & \quad s \leftarrow s + \frac{1}{6}f(x^-) + \frac{4}{6}f(x) + \frac{1}{6}f(x^+) \\
\quad & \text{end for} \\
\quad & s \leftarrow hs \\
\quad & \text{return } s
\end{align*}
\]

Algorithm 6.3 and a computer implementation that follows the algorithm line by line using similar symbols represent a safe development from description (6.3) of the numerical method. It should not be difficult to get things right and obtain working code.

Optimization of the Algorithm

The downside of Algorithm 6.3 is that we perform more evaluations of the function \(f(x)\) than necessary. Computing \(f(x)\) is probably the most expensive part of such algorithms, so avoiding too many function evaluations will have a significant impact on CPU time. Even very intelligent compilers will probably have a hard time detecting that we are performing too many function evaluations, so this is work for a human programmer.

The main observation is that we evaluate \(f(x_{i-1})\) and \(f(x_i)\) in the \(i\)th term of the sum in (6.3) and recalculate \(f(x_i)\) in the next term of the sum (where we need \(f(x_i)\) and \(f(x_{i+1})\)). We should avoid the recalculation of \(f(x_i)\). How to do this is
easy to see if we write out \( \sum_{i=1}^{n} \left( \frac{1}{6} f(x_{i-1}) + \frac{4}{6} f(x_{i\frac{1}{2}}) + \frac{1}{6} f(x_{i+1}) \right) \) as

\[
\begin{align*}
\frac{1}{6} f(x_0) + \frac{4}{6} f(x_{\frac{1}{2}}) + \frac{1}{6} f(x_1) + \\
\frac{1}{6} f(x_1) + \frac{4}{6} f(x_{\frac{3}{2}}) + \frac{1}{6} f(x_2) + \\
\frac{1}{6} f(x_2) + \frac{4}{6} f(x_{\frac{5}{2}}) + \frac{1}{6} f(x_3) + \\
\vdots + \\
\frac{1}{6} f(x_{n-1}) + \frac{4}{6} f(x_{n\frac{1}{2}}) + \frac{1}{6} f(x_n).
\end{align*}
\]

This expression can be expressed as two sums plus a contribution from the end points:

\[
\frac{1}{6} f(x_0) + \frac{4}{6} \sum_{i=1}^{n} f(x_{\frac{1}{2}}) + \frac{2}{6} \sum_{i=1}^{n-1} f(x_i) + \frac{1}{6} f(x_n).
\]

A more efficient formula for Simpson’s rule is therefore

\[
\int_a^b f(x) \, dx \approx \frac{1}{6} h \left( f(a) + f(b) + 2 \sum_{i=1}^{n-1} f(x_i) + 4 \sum_{i=1}^{n} f(x_{\frac{i}{2}}) \right). \tag{6.4}
\]

The corresponding algorithm appears in Algorithm 6.4. Of course, we could perform the hand optimization of incrementing \( x_i \) and \( x_{i\frac{1}{2}} \) instead of using \( a + i h \) and \( a + (i - \frac{1}{2})h \). This saving of one multiplication in each loop can be detected automatically by a compiler or simply drown in the work required by evaluating \( f(x) \).

**Algorithm 6.4**

*Optimized Simpson’s Rule.*

\begin{verbatim}
Simpson (a, b, f, n)
    h = \frac{b-a}{n}
    s_1 = 0
    for i = 1, \ldots, n - 1
        x = a + i h
        s_1 \leftarrow s_1 + f(x)
    end for
    s_2 = 0
    for i = 1, \ldots, n
        x = a + (i - \frac{1}{2}) h
        s_2 \leftarrow s_2 + f(x)
    end for
    s \leftarrow \frac{1}{6} h (f(a) + f(b) + 2s_1 + 4s_2)
    return s
\end{verbatim}
Algorithm 6.5

Simpson’s Rule with One Loop.

Simpson \((a, b, f, n)\)

\[
\begin{align*}
    h &= \frac{b-a}{n} \\
    s_1 &= 0 \\
    s_2 &= 0 \\
    \text{for } i = 1, \ldots, n \\
    &\quad \text{if } i < n \\
    &\quad \quad x = a + ih \\
    &\quad \quad s_1 \leftarrow s_1 + f(x) \\
    &\quad \text{end if} \\
    &\quad x = a + (i - \frac{1}{2})h \\
    &\quad s_2 \leftarrow s_2 + f(x) \\
    \text{end for} \\
    s &\leftarrow \frac{1}{6} h (f(a) + f(b) + 2s_1 + 4s_2) \\
\end{align*}
\]

return \(s\)

Compiler Optimization Issues

Why do we split the computations in Simpson’s rule into two sums, expressed as two loops in Algorithm 6.4? We could get away with one loop as demonstrated in Algorithm 6.5. The single loop now needs an if-test to avoid adding contributions to \(s_1\) when \(i = n\). Unfortunately, if-tests inside loops tend to reduce the compiler’s possibilities for optimization. Two plain loops are therefore normally much more efficient than one loop with an if-test. The loss of speed introduced by if-tests inside loops depends on the actual code, the programming language, the compiler, problem-dependent parameters, and so on. In the present case, one can argue that function calls also reduce compiler optimizations in the same way as an if-test, so Algorithm 6.5 may not be much less efficient than Algorithm 6.4. However, we can hardly avoid calling \(f(x)\) in numerical integration, but the if-test is easy to avoid. We should mention here that one way to avoid the if-test in Algorithm 6.5 is to remove the test and simply subtract the extra term \(f(a + nh)\) from \(s_1\).

Another aspect of the current discussion is that some smart compilers will avoid function calls as a part of the optimization; the body of the code for \(f(x)\) is inserted directly in the loop as an expression. For example, if we integrate \(f(x) = x \sin x\), a smart compiler can translate the code, e.g., in the following way:

\[
\begin{align*}
\text{... for } i = 1, \ldots, n \\
\text{...} \\
    x &= a + (i - \frac{1}{2})h \\
    s_2 &\leftarrow s_2 + 1 + x \sin x \\
\end{align*}
\]
Of course, hand coding of such an optimization would make the implementation valid for only a specific function. Calling $f(x)$ instead makes the implementation valid for any $f$.

Testing the computational efficiency of integrating $f_1(x) = e^{-x^2} \log(1 + x \sin x)$ and $f_2(x) = 1 + x$ shows that Algorithm 6.4 clearly runs faster than Algorithm 6.3. The difference depends on the amount of compiler optimization and the complexity of $f(x)$. With full optimization and $f_2(x)$, Algorithm 6.4 reduced the CPU time by about 20% in comparison with Algorithm 6.3. The theoretical saving is 33%, since we reduce the number of function evaluations from $3n$ to $2n + 1$.

Computer implementation and debugging can often be very time consuming. When preparing a problem in scientific computing for implementation, one should start with the simplest version of the algorithm. Smart rewriting and hand optimizations in the code can easily result in hunting errors for hours. On the other hand, training the ability to rewrite algorithms in more efficient forms is an important part of scientific computing. Real-life applications of numerical computing in science and technology make such harsh demands on computer power that serious work with optimization is crucial. The bottom line is that such optimization work should be performed after safe and easy-to-understand algorithms are implemented and thoroughly tested. Thinking “constantly” of optimization when carrying out the mathematics and programming has been a tradition in scientific computing, and we believe the result has been too many codes that compute wrong numbers simply because the “smart” expressions increased the complexity of the problem and the code at too early a stage in the development.

### 6.1.5 Adaptive Integration Rules

A basic problem when applying the trapezoidal or Simpson’s rule is to find a suitable value of $n$. Two factors influence the choice of $n$: the desired accuracy of the integral and the shape of $f$. A simple strategy is to apply the integration rule for a sequence of $n$ increasing values, say, $n_0, n_1, n_2, \ldots$, and stop when the difference between two consecutive integral values becomes negligible. This is called an adaptive algorithm, because the number of points will adapt to the desired accuracy and the shape of $f$.

Define $I(a, b, f, n)$ as an approximation to $\int_a^b f(x)dx$ using a specific numerical integration rule with $n$ points, and let $\epsilon$ be some specified tolerance. The following pseudo code expresses adaptive integration:

```plaintext
r = I(a, b, f, n_0)
for n = n_1, n_2, n_3, \ldots
  s = I(a, b, f, n)
  if |s - r| \leq \epsilon then
    return s
  else
    r = s
  end if
end for
```
A typical choice of $n_i$ is $(b - a)/2^i$. If $\epsilon$ is too small, the adaptive algorithm can run for an unacceptable long time (or forever if $\epsilon$ is so small that round-off errors destroy the convergence of the integral as $n$ grows). We should therefore return when $n$ exceeds a prescribed large value. Algorithm 6.6 lists the pseudo code.

**Algorithm 6.6**

Adaptive Integration: Repeated Evaluations.

```pseudo
adaptive_integration (a, b, f, I, \epsilon, n_{max})
  r = I(a, b, f, n_0)
  for n = n_1, n_2, n_3, \ldots
    s = I(a, b, f, n)
    if |s - r| \leq \epsilon then
      return s
    else
      r = s
    end if
    if n > n_{max} then
      print error message, return s
    end if
  end for
```

Another strategy for developing adaptive rules is to base the integration on an unequal spacing of the evaluation points $x_i$ in the interval $[a, b]$. In the case of the trapezoidal rule, we introduce evaluation points $x_i$, where $x_0 = a$, $x_n = b$, and $x_i < x_{i+1}$, $i = 0, n - 1$. The trapezoidal rule for unequal spacing can be expressed as

$$\int_a^b f(x)dx \approx \sum_{i=1}^{n} \frac{1}{2}(x_i - x_{i-1})(f(x_{i-1}) + f(x_i)). \quad (6.5)$$

This can be directly expressed in pseudo code. In case $f$ is expensive to evaluate, we could develop an optimized version where $f$ is evaluated only once at every point:

$$\int_a^b f(x)dx \approx \sum_{i=1}^{n} \frac{1}{2}(x_i - x_{i-1})(f(x_{i-1}) + f(x_i)) \quad (6.6)$$

$$= \cdots + \frac{1}{2}(x_i - x_{i-1})(f(x_{i-1}) + f(x_i)) +$$

$$\frac{1}{2}(x_{i+1} - x_i)(f(x_i) + f(x_{i+1})) + \cdots \quad (6.7)$$

$$\frac{1}{2}(x_{i+1} - x_i)(f(x_i) + f(x_{i+1})) + \cdots \quad (6.8)$$
The corresponding pseudo code is given as Algorithm 6.7. One still needs however, to devise a strategy for choosing the unequal spacing, i.e., determine the values of \( x_0, \ldots, x_n \). An intuitive approach is to cluster evaluation points where \( f(x) \) is rapidly varying, perhaps in a way that makes the error approximately the same on every interval \([x_{i-1}, x_i]\).

Algorithm 6.7

**Optimized Trapezoidal Integration: Unequal Spacing.**

```plaintext
trapezoidal (a, b, f, x_0, \ldots, x_n)

s = 0
for i = 1, \ldots, n - 1
    s ← s + f(x_i)(x_{i+1} - x_{i-1})
end for
s ← s + f(a)(x_1 - x_0) + f(b)(x_n - x_{n-1})
return 0.5 \cdot s
```

The error when applying the trapezoidal rule on an interval \([x_{i-1}, x_i]\) can be shown to be

\[
\hat{E} = -\frac{1}{12} (x_i - x_{i-1})^3 f''(\xi),
\]

where \( \xi \) is some point in \([x_{i-1}, x_i]\). Using a finite difference to approximate \( f'' \) at the midpoint of the interval,

\[
f''(\frac{1}{2} (x_{i-1} + x_i)) \approx \frac{1}{4(x_i - x_{i-1})^2} \left( f(x_{i-1}) - 2f(\frac{1}{2} (x_{i-1} + x_i)) + f(x_i) \right),
\]

we can replace (6.11) by an approximate error quantity

\[
E(x_{i-1}, x_i, f) = -\frac{x_i - x_{i-1}}{48} \left( f(x_{i-1}) - 2f(\frac{1}{2} (x_{i-1} + x_i)) + f(x_i) \right).
\]

We could now start with an equally spaced coarse distribution \( x_i = a + ih \), compute the error in each interval, and divide an interval into two new equally spaced segments if the error is greater than \( \epsilon \), where \( \epsilon \) is the largest acceptable error on an interval. This procedure can be repeated until no more refinement is necessary. Of course, we should also stop the refinement if the number of points \( x_i \) becomes unacceptably large. Algorithm 6.8 presents the pseudo code for one level of refinement. We provide \( x_0, \ldots, x_n \) as input and obtain a new set of points \( y_0, y_1, y_2, \ldots \), where each original interval is either preserved or divided into two new intervals.
Algorithm 6.8

*Interval Refinement based on Error Estimate.*

\[
\text{refine1 } (a, b, f, E, \epsilon, n_{\text{max}}, x_0, \ldots, x_n) \\
\quad j = 0 \\
\quad \text{for } i = 1, \ldots, n \\
\quad \quad y_j = x_{i-1}; \; j \leftarrow j + 1 \\
\quad \quad e = |E(x_{i-1}, x_i, f)| \\
\quad \quad \text{if } e > \epsilon \text{ then} \\
\quad \quad \quad \text{if } j > n_{\text{max}} - 2 \text{ then} \\
\quad \quad \quad \quad \text{error: too many refinements} \\
\quad \quad \quad \text{end if} \\
\quad \quad \quad y_j = \frac{1}{2}(x_{i-1} + x_i); \; j \leftarrow j + 1 \\
\quad \quad \text{end if} \\
\quad y_j = x_n \\
\quad \text{end for} \\
\quad \text{return } y_0, \ldots, y_j
\]

What we want is actually a *recursive* procedure; we want to call Algorithm 6.8 repeatedly until all interval error estimates are less than \(\epsilon\) or until we reach \(n_{\text{max}}\) points. The recursive version appears as Algorithm 6.9. We notice that the algorithm is applicable to any integration rule as long as we provide a function \(E\) for computing the error in an interval. If you are not familiar with recursive algorithms, you should definitely work out Exercise 6.5.

Algorithm 6.9

*Recursive Interval Refinement based on Error Estimate.*

\[
\text{refine } (a, b, f, E, \epsilon, n_{\text{max}}, x_0, \ldots, x_n) \\
\quad j = 0 \\
\quad \text{refined} = \text{false} \\
\quad \text{stop} = \text{false} \\
\quad \text{for } i = 1, \ldots, n \\
\quad \quad y_j = x_{i-1}; \; j \leftarrow j + 1 \\
\quad \quad e = |E(x_{i-1}, x_i, f)| \\
\quad \quad \text{if } e > \epsilon \text{ then} \\
\quad \quad \quad \text{if } j > n_{\text{max}} - 2 \text{ then} \\
\quad \quad \quad \quad \text{stop} = \text{true}; \text{jump out of } i\text{-loop} \\
\quad \quad \quad \text{end if} \\
\quad \quad \quad \text{refined} = \text{true} \\
\quad \quad \quad y_j = \frac{1}{2}(x_{i-1} + x_i); \; j \leftarrow j + 1 \\
\quad \quad \text{end for} \\
\quad y_j = x_n \\
\quad \text{if stop or not refined then} \\
\quad \quad \text{return } y_0, \ldots, y_j \\
\quad \text{else} \\
\quad \quad \text{refine } (a, b, f, \epsilon, n_{\text{max}}, y_0, y_1, \ldots, y_j)
\]
6.1 Algorithms Expressed as Pseudo Code

6.1.6 Ordinary Differential Equations

Writing algorithms in mathematical pseudo code represents a considerable portion of the software development work, at least for the (quite simple) type of problems we deal with in the present book. Before looking at the translation of pseudo code to specific computer languages it can therefore be instructive to go through another example on pseudo code development. Consider an ODE,

\[ u'(t) = f(u), \quad u(0) = U_0. \]

solved approximately by Heun’s method:

\[ u_{n+1} = u_n + \frac{\Delta t}{2} [f(u_n) + f(u_n + \Delta t f(u_n))]. \quad (6.13) \]

Here, \( u_n \) is an approximation to the exact solution evaluated at the time point \( n\Delta t \).

A direct translation of (6.13) to an algorithmic form leads to Algorithm 6.10.

**Algorithm 6.10**

```
Heun's Method.
heun ( f, U0, Δt, N )
    u0 = U0
    for n = 0, . . . , N − 1
        un+1 = un + Δt/2 [f(un) + f(un + Δt f(un))]
    end for
    return uN
```

We have previously mentioned that mathematical symbols with indices normally translate to arrays in a program, which in the present case means that we store \( u_1, u_2, \ldots, u_N \). An optimization-oriented person who is very concerned about reducing computer memory might claim that to compute the return value \( u_N \), we can get away with storing only one \( u \) value at a time, as expressed in Algorithm 6.10.

**Algorithm 6.11**

```
Memory-Optimized Heun's Method.
heun ( f, U0, Δt, N )
    u = U0
    for n = 0, . . . , N − 1
        u ← u + Δt/2 [f(u) + f(u + Δt f(u))]
    end for
    return u
```
Computing \( u \) at some given time \( t_k = k \Delta t \) is now a matter of just calling \( \text{heun}(f, U_0, \Delta t, k) \). Suppose we want to plot \( u(t) \) as a function of \( t \) for \( 0 \leq t \leq T \), using \( N \) points, we can simply implement the following pseudo code:

\[
\Delta t = \frac{T}{N} \\
\text{for } i = 0, \ldots, N \\
\quad u_i \leftarrow \text{heun}(f, U_0, \Delta t, i) \text{ (Algorithm 6.11)} \\
\quad \text{write } (i \Delta t, u_i) \text{ to file} \\
\text{end for}
\]

The result is typically a two-column file, with \( t \) and \( u \) values in columns 1 and 2, respectively, which can be plotted by (probably) any plotting program.

The outlined implementation is very inefficient, yet common among newcomers to scientific computing. The implementation is very efficient in reducing the need for computer memory, unfortunately at the significant cost of CPU time: Every time we call the \( \text{heun}(f, U_0, \Delta t, i) \) function, we recompute what we did in the previous call, plus compute a single new \( u \) value. We should, of course, integrate the differential equations up to time \( T \) and return all the \( u_1, \ldots, u_N \) values computed in Algorithm 6.10. After the call to the heun function we can dump the returned array to a file for plotting and data analysis.

Looking more closely at Algorithm 6.10, we realize that the function evaluation \( f(u_n) \) is carried out twice. We should avoid this. A more efficient computational approach, which should be considered by the reader for implementation, appears in Algorithm 6.12. This implementation is flexible; we can call it to advance the solution one time step at a time, we can compute the complete solution up to the desired final point of time, or we can call the function several times, computing a chunk of \( u_n \) values in each call.

**Algorithm 6.12**

*Optimized Heun’s Method.*

\[
\text{heun}(f, U_0, \Delta t, N) \\
\quad u_0 = U_0 \\
\quad \text{for } n = 0, \ldots, N - 1 \\
\quad \quad v = f(u_n) \\
\quad \quad u_{n+1} = u_n + \frac{\Delta t}{2}[v + f(u_n + \Delta t v)] \\
\quad \text{end for} \\
\quad \text{return } u_0, u_1, \ldots, u_N
\]

6.2 About Programming Languages

There are hundreds of programming languages. Some of them are well suited for scientific computing, others are not. We will present the characteristics of the most popular languages for scientific computing. Our aim is to demonstrate that as soon
as you have expressed an algorithm as mathematical pseudo code, the step to a real
programming language is quite small. The implementation of an algorithm in a par-
ticular language will of course depend on the syntax and built-in functionality of that
language. Even for the simple algorithms from Sect. 6.1, the realizations of the algo-
rithms in different languages (see Sects. 6.3 and 6.4) vary considerably. Deciding
upon the most convenient programming tool is greatly subject to personal taste and
experience. Nevertheless, modern computational scientists need to master several
programming languages and have the ability to quickly jump into a new language.
The reason is obvious: Different tools have different strengths and weaknesses and
hence are suitable for different type of problems.

We shall discuss four basic issues that influence the choice of programming
language for scientific computing tasks:

– Static typing versus dynamic typing
– Computational efficiency
– Built-in numerical high-performance utilities
– Support for user-defined objects

The languages we bring into the discussion are Fortran 77, C, C++, Java, Maple,
Matlab, and Python. All of these languages are widely used in scientific computing.
There are different versions of Fortran, e.g., Fortran 95 and Fortran 2000, which
have some of the features of C++, Java, and Python. Fortran 77, on the other hand,
is a small and compact language with fewer features than C. We will normally spec-
ify the type of Fortran language, and use the term Fortran (without a number) when
we speak of the Fortran family of languages in general.

6.2.1 Static Typing Versus Dynamic Typing

Computer languages such as Fortran, C, C++, and Java are said to be strongly
typed, or statically typed. This means that the programmer must explicitly
write the type of each variable. If you want to assign the value 3 to a variable called
a, you must first specify the type of a. In this case a can be an integer or a real (single
precision or double precision) variable, since both types can hold the number 3. Say
you specify a to be an integer. Later on in the program you cannot store a string in
a, because a can only hold an integer; a is related to a memory segment consisting of
four bytes, which are sufficient to store an integer, but too large or (more likely) too
small to store a string. Even if we could store a four-character string in four bytes,
the program would interpret these four bytes as an integer, not as four characters.

Maple, Matlab, and Python are examples of programming languages where one
does not need to specify the type of variables. If we want to store the number 3
in a variable a, we usually write a = 3 (or a := 3 in Maple) to perform the assign-
ment. We may well store a string in a afterward, and then a real number, if desired.
The variable a will know its content, i.e., it will know whether it contains an inte-
ger, a real number, or a string and adjust its behavior accordingly. For example, if
we multiply \( a \) by 15, this is a legal mathematical operation, provided \( a \) is a number; otherwise \( a \) will know that it does not make sense for it to participate in a multiplication. The type information store in \( a \) is there, but we can say that the typing is dynamic, in the sense that the programmer does not explicitly write the type of \( a \) and \( a \) can dynamically change the type of its content during program execution. Dynamic typing gives great flexibility and a syntax that is closer to the mathematical notation of our pseudo code language.

Static typing is often said to be very useful for novice programmers. Incompatible types are quickly detected and reported before the program can be run. Suppose you hit the \( \texttt{s} \) key instead of the \( \texttt{a} \) key when writing the statement \( a = 3 \). If you do not introduce a variable \( s \) elsewhere, \( s \) is not declared with a type, and a typing error is detected. With dynamic typing nothing about types is known before the program is run. Incompatible types are detected when operations become illegal, but a writing error such as \( s = 3 \) instead of \( a = 3 \) is not detected; \( s \) is as good as any variable name, and it is brought into play by the assignment. However, as soon as the program works with \( a \), this variable is now not initialized, and an error will result.

The authors have programmed in languages with both static and dynamic typing. The conclusion is that the debugging styles are a bit different, but we cannot claim that one of the approaches generally leads to fewer errors than the other.

### 6.2.2 Computational Efficiency

Some computer languages are faster than others. For example, the loop over \( i \) in Algorithm 6.4 will be much faster in Fortran and C/C++ than in Maple, Matlab, or Python, no matter how much we try to optimize the implementation. The reason that Fortran, C, and C++ are so much faster stems from the fact that these languages are compiled into machine code. We can classify computer languages as either interpreted or compiled. This distinction reflects the potential speed of the language and is therefore of particular interest in scientific computing. Normally, compiled languages have static typing, whereas interpreted languages have dynamic typing. Java is something in between: It has static typing, was originally interpreted, but is being equipped with compiler techniques.

Fortran, C, and C++ are examples of compiled languages. A compiler translates the computer program into machine code, i.e., low-level, primitive instructions tied to the hardware. Since most programs call up functionality in some external libraries, the machine code of a program must be linked with a machine code version of the required libraries. Compilation and linking are normally two distinct steps. The machine code of the program and the libraries are merged in one file, the executable. To run the program, you simply write the name of the executable.

Maple, Matlab, and Python are examples of interpreted languages. A program is read (line by line) by an interpreter that translates statements in the code into function calls in a library. The translation takes place while the program is running. Hence, inside a loop the translation of a particular statement is repeated as many
times as iterated in the loop. This causes the program to run much more slowly than in the case where we could just execute pre-made machine instructions. The benefit is that we get a high degree of flexibility; statements and data structures can change dynamically at run time in a safe and convenient way from a user’s point of view. Interactive computing is one application of this flexibility.

Comparing implementations of Algorithm 6.4 in different languages shows that the speed of the various programs varies greatly. Using compiled languages results in significantly faster programs than interpreted languages. Some numerical applications require so much computing power that one is forced to use the fastest algorithms and languages in order to solve the problem at hand. Other applications demand only minutes or hours of execution time on a desktop computer, and in these cases one can often trade computational efficiency in favor of increased human efficiency, i.e., quicker, safer and more convenient programming.

### 6.2.3 Built-in High-Performance Utilities

Programming in Fortran, C, C++, and Java can be very different from programming in Maple, Matlab, and Python, especially when it comes to algorithms with loops. In the former four languages you can implement a loop as in Algorithm 6.4 using basic loop constructions in those languages, and the performance will be very good. This means that there will often be small differences between your implementation and a professional implementation of the algorithm made available through a numerical library. Many researchers and engineers like to create their own implementation of algorithms to have full control of what is going on. This strategy often leads to only a minor loss of performance.

The situation is different in Maple and Python, and also to some extent in Matlab. Plain loops in these languages normally lead to slow code. A Maple implementation of Algorithm 6.4 can easily be more than 150 times slower than a similar implementation in Fortran 77. However, this comparison is not fair. Plain, long loops in Maple are to some extent a misuse of the language, especially if we try to implement algorithms that are available in libraries or as a part of the language. Numerical integration is an example; Maple has a rich functionality for numerical integration, where accuracy and performance can be controlled. Calling up a built-in numerical integration rule in Maple (which means an algorithm of much higher sophistication than the trapezoidal rule) produces a result in about the same time as required by our hand-made Fortran 77 function. In other words, you should not consider implementation of basic numerical algorithms in Maple, because this will give low performance; you should utilize built-in functionality. Clearly, not all algorithms are available in Maple, and sometimes you have to make your own implementations. It is then important to try to break up the steps in an algorithm into smaller steps, which may be supported by built-in functionality.

The comments about Maple are also relevant for Matlab and Python. All these three programming platforms have a large collection of highly optimized functions
in numerical libraries. Algorithms to be implemented in Maple, Matlab, and Python should utilize the languages’ high-performance built-in functionality to as high a degree as possible. Hence, programmers using these languages need to be familiar with the libraries, otherwise they will both reinvent the wheel (by re-implementing basic algorithms) and decrease performance. While loops play an important role in compiled languages, loops are very slow in interpreted languages and should hence be avoided. Instead, one needs to express the algorithms in terms of built-in basic vector operations, where all loops are solely executed in libraries written in compiled languages. The process of rewriting an algorithm in terms of basic vector operations is usually referred to as vectorization and constitutes the subject of Sect. 6.3.7.

We should mention that Matlab has a just-in-time compiler that can automatically turn loops over arrays into as efficient code as vectorization can offer. However, the just-in-time compilation strategy works only in certain situations, and in particular not if there are function calls inside the loops, as we have in the numerical integration algorithms.

6.2.4 Support for User-Defined Objects

Fortran 77 has only a few different types of variables: integer, real, array, complex, and character string. One can build any code with these types of variables, but as soon as the complexity of the program grows, one may want to group a collection of primitive variables into a new, more advanced variable type. Many languages therefore allow the programmer to define new variable types. In C, one can define a `struct` to be a collection of basic C variables and previously defined `struct`s. This construction acts as a method for creating user-defined variable types.

When people start to create their own types with `struct`s and similar constructions in other languages, it becomes apparent that a user-defined variable should not only store information, but also be able to manipulate and process that information. This results in more intelligent variable types, but requires the `struct` to contain functions as well as data. C++ is an extension of C that offers user-defined variable types containing both functions and data. Such a user-defined type is normally referred to as a `class`. Fortran 90 and Fortran 95 are extensions of Fortran 77 (and earlier versions known as Fortran IV and Fortran 66) where the programmer can collect data and functions in `modules`. Modules can do many of the same things as classes, but classes in C++ are much more advanced than the modules in Fortran 90/95. The Fortran 2000 initiative aims at equipping Fortran with modules having the functionality of classes that programmers expect today.

Java supports classes, but the flexibility of the classes is less than in C++. On the other hand, Java has functionality that makes the language much more user-friendly and easier to program with than C++. Python has classes that are more advanced than those in C++. Matlab originally had only matrices and strings, but recent extensions to Matlab allow the definition of classes. Maple has many variable
types, but no way to create user-defined types. Having said this, we should add that languages with a rich collection of built-in data types and dynamic typing, such as Maple and Python, often offer the ability to create heterogeneous lists or hash structures that replace the traditional use of classes in C++ and Java.

A class can be viewed as a definition of a new type of variable. To create variables of this type, one creates objects of the class. The program will therefore work with objects in addition to variables of built-in types. Programming with objects is a bit different from programming with fundamental built-in types, such as real, integer, and string. In the latter case, collections of primitive variables are shuffled in and out of functions (subprograms) and the software is organized as a library of functions. With classes, the functions are often built into the classes, and the software is a collection of classes (i.e. data types) instead of a function library.

The real power of class-based programming comes into play when classes can be related to each other in a kind of family tree, where children classes inherit functionality and data structures from parent classes. Although the implementational and functional details can differ between members in such a tree of classes, one can often hide these differing details and work solely with the topmost parent class. This is called object-oriented programming and constitutes a very important programming technique in computer science. The Java language is very tightly connected to object-oriented programming. Object-oriented programming is highly valuable in scientific computing, especially for administering large and complicated codes, but the traditionally popular languages Fortran and C do not support this style of programming.

6.3 The Trapezoidal Rule in Different Languages

On the following pages we shall give a glimpse of some widely used programming languages: Fortran 77, C, C++, Java, Matlab, Maple, and Python. All three languages are popular in scientific computing communities. The exposition is strongly example-oriented. The simple trapezoidal rule from Algorithm 6.2 on page 201 is implemented in each of the mentioned languages. The code segments are briefly explained and accompanied by some comments related to topics such as static typing and computational efficiency. In Sect. 6.4 we shall implement Heun’s method from Algorithm 6.12 in the aforementioned computer languages and thereby learn more about arrays and file handling in different languages. When solving ODEs, we also need to display the solution graphically, and this is shown by plotting the solution using Gnuplot and Matlab.

6.3.1 Code Structure

The main structure of the code for performing numerical integration with the aid of the trapezoidal rule will be quite independent of the programming tool being
utilized. Our mathematical model is an integral,

\[ I = \int_a^b f(x) \, dx, \]

and the purpose is to compute an approximation of \( I \). In a program we need to

1. Initialize input data: \( a, b, \) and \( n \)
2. Specify a function \( f(x) \)
3. Call a program module that implements Algorithm 6.2
4. Write out the value of \( I \)

The input data are normally provided by the user of the program at run time, but
for simplicity, we shall explicitly set \( a = 0, b = 2, \) and \( n = 1,000 \) in our sample
codes.

The function \( f(x) \) is selected as \( e^{-x^2} \log(1 + x \sin x) \) and implemented with
the name \( f1 \). Most codes also have a simple linear function \( 1 + x \), called \( f2 \) and
used for checking that the numerical results are correct. Since \( 1 + x \) is integrated
without numerical errors, when using the trapezoidal rule, we should expect \( I = \int_0^2 (1 + x) \, dx = 4 \) to machine precision. If the program works for \( f2 \), we expect it
to work for \( f1 \).

Algorithm 6.2 is more or less directly translated to a function. The main differ-
ence between the various implementations is how \( f(x) \) is technically treated as an
argument in this function.

### 6.3.2 Fortran 77

The Fortran language was designed in the 1950s as a high-level alternative to assem-
bly programming. With Fortran, scientists could express mathematical formulas in a
program using a syntax close to the syntax of mathematics. Since the 1950s, Fortran
has been improved several times, resulting in Fortran IV, Fortran 66, Fortran 77,
Fortran 90, Fortran 95, and Fortran 2000. The latter three versions are very different
from the former ones, so it makes sense to talk about two families of Fortran, before
and after Fortran 90.

Fortran 77 is a quite primitive language in the sense that there are few variable
types (only integer, real, complex, array, and string), few keywords, few control
structures, and a strict layout of the code. These limitations are not significant until
one tries to build fairly large codes. Fortran IV and Fortran 66 are even more primi-
tive and are of no significance today, except that there is lots of legacy code written
in these languages.

Fortran 77 is without competition the most widespread programming language
for scientific computing applications. The reasons for this are simplicity, tradition,
and high performance. The newer versions, Fortran 90, 95, and 2000, support pro-
gramming with objects (called modules), and have modernized Fortran, to some
extent, in the direction of C++ and Java. However, Fortran 77 is normally faster than the newer versions and therefore preferred by many computational scientists.

Fortran 77 is very well suited for implementing algorithms with CPU-intensive loops, especially when the loops traverse array structures. Applications involving sophisticated data structures, text processing, or large amounts of code will normally benefit from being implemented in more modern languages such as C++ or Fortran 90/95/2000.

Algorithm 6.2 can be implemented in a Fortran 77 function as illustrated below:

```fortran
real*8 function trapezoidal (a, b, f, n)
real*8 a, b, f
external f
integer n
real*8 s, h, x
integer i
h = (b-a)/float(n)
s = 0
x = a
do i = 1, n-1
   x = x + h
   s = s + f(x)
end do
s = 0.5*(f(a) + f(b)) + s
trapezoidal = h*s
return
end
```

Fortran 77 has some limitations on the layout of the statements, which seem a bit old-fashioned now:

- No statement must begin before column 7.
- There can only be one statement per line.
- Comments start in the first column, usually with the character `C`, and extend to the end of the line.

Some other rules are as follows:

- Fortran is case-insensitive, so whether you write `trapezoidal` or `TraPEZoidal` does not matter.
- All variables, i.e., arguments and local variables, must be declared before the first computational statement in the function.
- Real variables can be of different lengths; `real*4` denotes a four-byte single-precision variable, and `real*8` denotes an eight-byte double-precision variable.
- For loops are implemented with the `do–end do` construction (and hence are referred to as “do loops”).
- The `external` keyword is used to indicate an external function to be called.

Note that we compute $\frac{b-a}{n}$ by converting $n$ to a real variable (the `float` function performs this operation in Fortran). The reason is because division by an integer
will in some languages imply integer division, which yields $h = 0$ in the present case. It is a good habit to always convert an integer in division operations to real, regardless of how the computer language handles the division operation.

A test program calling up the trapezoidal function can look like the following:

```c
C test function to integrate:
real*8 function f1 (x)
real*8 x
f1 = exp(-x*x)*log(1+x*sin(x))
return
end

C main program:
program integration
integer n
real*8 a, b, result
external f1
a = 0
b = 2
n = 1000
result = trapezoidal (a, b, f1, n)
write (*,*) result
end
```

Compiling, Linking, and Executing the Program

Suppose the code segments above are stored in a file `int.f`. Fortran programs must be compiled and linked. On a Unix system this can take the form

```bash
unix> f77 -O3 -c int.f # compilation
unix> f77 -o int int.o # linking
```

The compilation step translates `int.f` into object (machine) code. The resulting file has the name `int.o`. The linking step combines `int.o` with the standard libraries of Fortran to form an executable program `int`. The `-O3` option means optimization level 3 when compiling the code. What a certain optimization level means depends on the compiler. On Linux systems the common Fortran compiler is the `g77` GNU compiler, not `f77`. You will normally find `g77` on other Unix systems too.

To run the program, we write the name of the executable, in this case `int`. On Unix systems you probably need to write `. /int`, unless your `PATH` environment variable contains a dot (the current working directory).

---

5 The result of dividing the integer $p$ by the integer $q$ is the largest integer $r$ such that $rq \leq p$. Note that if $q > p$, $r = 0$.

6 There is a significant security risk in having a dot in the `PATH` variable, so this is not recommended.
Measuring the CPU Time

The forthcoming subsections contain implementations of the trapezoidal algorithm in many other languages. The relative computational efficiency of these implementations is of significant interest. The CPU time consumed by an algorithm is a good measure of computational efficiency. Unix systems have a command `time` that can be used to infer the timings of a program. In the present case one can write

```
unix> time ./int
```

and the output will be something like

```
real 0m4.367s
user 0m4.310s
sys 0m0.050s
```

“Time” on a computer is not a unique term. The time it takes for the `int` program to execute, measured on a wall clock or wrist watch, is called `wall clock time`. Sometimes this time measurement is called `elapsed time` or `real time`. This measure of time may not reflect the efficiency of the program well if there are many other processes and users running concurrently on the machine. Therefore, other time measurements are introduced. The `user time` is the time it takes to execute the statements in the program, minus the time that is spent in the operating system on, e.g., reading and writing files. The time consumed by the latter operations is called `system time`. The sum of the user and system time constitutes the `CPU time`, i.e., the time it takes to execute all statements in the program.\(^7\) In the example above, the CPU time is \(4.310 + 0.05 = 4.360\) s. The accuracy of such timings varies with the computer system. To be on the safe side, the program should run for several seconds (say, at least 5 s).

The current Fortran 77 program with 10,000 repetitive calls to the function `trapezoidal`, with \(n = 1,000\), requires about 4 s on an IBM X30 laptop running Linux and using the `g77` compiler. (Note that a “smart” compiler may be so smart that it throws away repetitive calls unless we insert statements that require all the calls!) For comparison purposes it is advantageous to use relative time measures. We will hence report the timings of the other implementations as the actual CPU time divided by the CPU time required by the Fortran 77 code.

Remark

The CPU time spent by a set of statements can in most languages also be measured inside the program by a suitable function call. Unfortunately, Fortran has no unified standard for doing this, although many Fortran compilers offer a timing function (\(g77\) offers `dtime`). We therefore rely on the Unix command `time` for timing the complete execution of a program.

\(^7\) This is almost correct: Child processes, typically launching a new program inside another program, are not included in the CPU time, but this is of no relevance in the present simple program.
6.3.3 C and C++

The C language was invented in the beginning of the 1970s. Its main application was implementation of the Unix operating system. The combination of Unix and C became a dominating programming environment in the late 1980s. C is clearly a more advanced programming language than Fortran 77, and some computational scientists switched from Fortran 77 to C. C gives the program developer more flexibility, normally at the cost of a slight loss in computational performance. C has the same fundamental data types as Fortran 77, but offers a struct type, which can hold any composition of fundamental data types and other structs. This offers the possibility of creating more advanced data types than in Fortran 77.

C++ was designed to offer classes and object-oriented programming as in the famous SIMULA 67 language, but with C syntax and C as a subset. Several features of languages with classes were removed to improve performance. Classes in C++ extend C’s struct such that user-defined types contain both data and functions operating on the data. This provides much more flexible data types than the struct in C. Applications that can benefit from sophisticated user-defined data structures, and perhaps also the style of object-oriented programming, will normally be much simpler to implement in C++ than in Fortran 77 or C. The applications will also often be simpler to maintain and extend. However, C++ code can seldom compete with Fortran 77 when it comes to computational speed, although the difference is small. With some tweaking, C and C++ can on many occasions be as fast as, or even slightly faster, than Fortran 77, but this depends on the type of algorithms and the involved data structures.

C and C++ programs for Algorithm 6.2 look very similar since this is a very simple problem. C++ has a slightly more readable syntax, so we present the algorithm in that language first:

```c
typedef double (*fptr) (double x);

double Trapezoidal (double a, double b, fptr f, int n)
{
    double h = (b-a)/double(n); double s = 0; double x = a;
    for (int i = 1; i <= n-1; i++) {
        x = x + h;
        s = s + f(x);
    }
    s = 0.5*(f(a) + f(b)) + s;
    return h*s;
}
```

C and C++ have, unfortunately, some constructs that make it somewhat difficult to explain even very simple programs to the novice. For example, the `Trapezoidal` function needs a function `f(x)` as an argument, and this argument must be a function pointer, here called `fptr` and defined through a `typedef` statement. The rest of the code should be easier to understand. Real variables with double precision (real*8 in Fortran 77 terminology) are called `double` in C and C++. The `Trapezoidal` function returns a `double` (the approximation to the integral). Contrary to Fortran 77, the type of each argument precedes the argument name, providing a more compact
code. Every statement ends with a semi-colon, and one can have as many statements as desired on the same line. In C++, variables can be declared wherever they are needed. The loop is implemented via the for construction, which is adopted in a wide range of programming languages.

A test program, calling up the Trapezoidal function to perform the numerical integration of a specific function, can take the form

```cpp
#include <iostream>
#include <cmath>

/* the Trapezoidal function: */
...

/* test function to be integrated */
double f1 (double x)
{
    return exp(-x*x)*log(1+x*sin(x));
}

int main() // main program
{
    double a = 0, b = 2; int n = 1000;
    double result = Trapezoidal (a, b, f1, n);
    std::cout << result << std::endl;
}
```

The #include statements include information about functions in libraries (here input/output functionality from iostream and mathematical functions from cmath); the compiler needs to see a declaration of a function, i.e. the type of return value and the type of arguments, before a call to the function can be compiled. For example, calling the exponential math function exp requires us to include the cmath file where exp is declared. If you call the function with the wrong types of arguments, a compiler error will be issued. Fortran 77 has no information about the functions one calls, so if you provide the wrong type or wrong number of arguments, the call will compile normally, but the run-time behavior is unpredictable. Very often strange things happen, and it requires some experience to find such bugs. Many programmers switched from Fortran 77 to C or C++ just because the C and C++ compilers find so many more coding errors.

The code segments above are written in C++, but the C version is quite similar. Since C++ has C as a subset and is a more flexible computer language, the authors recommend using C++ instead of plain C. Some constructs in C++, especially related to programming with classes, can result in a slight performance loss. One can, nevertheless, often rewrite time-critical parts in C++ programs using the C subset in order to take advantage of the performance of C.

A very popular C compiler is gcc (GNU’s C compiler). On most Unix systems, the vendor’s compiler has the name cc. The C++ counterpart to gcc is g++. This is a good all-round compiler, but some commercial compilers, especially the KCC compiler, can give significantly better performance in scientific computing applications.
A typical session for compiling and linking a C++ program in a file \texttt{int.cpp} on a Unix machine, using the \texttt{g++} compiler, goes like this:

\begin{verbatim}
unix> g++ -O3 -c int.cpp  # compiling int.cpp to int.o
unix> g++ -o int int.o   # linking; form executable int
\end{verbatim}

These steps are the same as for Fortran 77 and will be similar for a C program; only the name of the compiler differs. Executing the program follows the description for the corresponding Fortran 77 program; just name the executable file.

The CPU time can be measured in C and C++ programs by calling the \texttt{clock} function. We have not done this. Instead we rely on the Unix \texttt{time} command. The performance test, involving calling the \texttt{Trapezoidal} function 10,000 times with \( n = 1,000 \), ran at the same speed as our Fortran 77 code. Normally, C and C++ run at the same or slightly lower speed than Fortran 77.

### 6.3.4 Java

Java was constructed by Sun Microsystems in the beginning of the 1990s and can be viewed as a simpler and more user-friendly version of C++. At the time Java was released, C++ had recently become a very popular and dominating language, but Java appeared to be more convenient and productive, so many programmers converted quickly to Java. Java is now the most popular programming language in the world.

Java programs are not compiled in the classic sense of Fortran, C, and C++ so performance was a significant negative feature for computational scientists. Despite numerous efforts to improve performance (and it is now indeed possible to obtain execution times close to those of C++ in some applications), the initial interest in Java as a language for scientific computing has declined. However, Java has become the dominant language for teaching introductory computer programming, so in the early stages of student programs it makes great sense to use Java for numerical computing. In the professional world of scientific computing, Fortran 77 dominates, but C++ and Matlab have a significant share.

Let us look at a Java implementation of Algorithm 6.2. At first sight, this implementation is significantly less straightforward than the implementations in Fortran or C/C++. The reason is because one has to work with classes, because all functions in Java must belong to a class. The bottom line is, therefore, that we implement the functions from our Fortran and C/C++ programs as functions in otherwise empty Java classes. We also note that functions in Java are referred to as \texttt{methods}.

\begin{verbatim}
import java.lang.*;

interface Func { // base class for functions f(x)
   public double f (double x); // default (empty) impl.
}

class Trapezoidal {
   public static double integrate (double a, double b,
                                  Func f, int n)
\end{verbatim}
```java
double h = (b-a)/((double)n);
double s = 0;
double x = a;
int i;
for (i = 1; i <= n-1; i++) {
    x = x + h;
    s = s + f.f(x);
}
s = 0.5*(f.f(a) + f.f(b)) + s;
return h*s;
}
```

The `integrate` method is very similar to Algorithm 6.2 and the implementations in most other languages. Java is statically typed and much of the syntax is inspired by C and C++. Calling `f(x)` is a bit different than in Fortran and C/C++, because we need to pass `f` as an object of type `Func` to class `Trapezoidal`'s `integrate` method. To evaluate `f`, we call the `f` method in class `Func`. An actual function to be integrated must be implemented as a method `f` in a subclass of `Func`. Here is an example:

```java
class f1 implements Func {
    public double f (double x) {
        return Math.exp(-x*x)*Math.log(1+x*Math.sin(x));
    }
}
```

That is, to send a function `f(x)` as an argument we actually need to implement the function as a class and make use of object-oriented programming. Many readers will find the Java code unnecessarily comprehensive in simple numerical problems because of this fact.

A main program in Java is actually a method `main` in some class, here a test class called `Demo`:

```java
class Demo {
    public static void main (String argv[]) {
        double a = 0;
        double b = 2;
        int n = 1000;
        double result = 0;
        int i;
        f1 f = new f1();
        //f2 f = new f2();
        for (i = 1; i <= 10000; i++) {
            result = Trapezoidal.integrate(a, b, f, n);
        }
        System.out.println(result);
    }
}
```

Since `integrate` is a static method in the `Trapezoidal` class, we can call the method without creating an object of type `Trapezoidal`. To pass the `f(x)` function defined in the `f` method in class `f1`, we must create an object of type `f1` and pass it to the `integrate` method. When developing larger computer applications, the idea of encapsulating everything in classes helps to modularize the code and ease
extensions, but in this simple example it may appear as unnecessary “overhead” compared to most of the other programming languages we address.

Suppose the shown code segments are located in a file `Trapezoidal.java` (the `.java` extension is required). The programs must first be compiled to bytecode:

```
unix> javac Trapezoidal.java
```

Now we can run the code by specifying the name of the class containing the main program, i.e., the `main` method:

```
unix> java Demo
```

There is unfortunately no built-in method for measuring the CPU time in Java programs so the easiest way to measure the efficiency is to use the Unix `time` command (`time java Demo`).

The CPU time ratio relative to the Fortran 77 and C/C++ codes was 2.1, i.e., Java ran at approximately half the speed of Fortran 77 and C/C++ in this example. The test was performed with Java Development Kit 1.1 for Linux. The speed of Java is likely to improve in future releases.

### 6.3.5 Matlab

Matlab was originally a user-friendly front end to efficient Fortran 77 libraries for numerical linear algebra computing. During the 1980s and especially the 1990s the Matlab environment became more and more user-friendly, and manifested Matlab as a leading development platform for at least simpler scientific computing applications. Working with Matlab is not too different from programming in Fortran 77, but some differences are striking:

- Statements can be issued in an interactive way, i.e., you can type a command and immediately view the result.
- There is no need to declare variables (what we called dynamic typing in a previous discussion).
- Matlab has many high-level commands that replace the need for writing detailed loops.
- If statements go wrong, Matlab normally issues easy-to-understand error messages.
- Matlab offers advanced, integrated visualization. This makes it easy to compute something and immediately display the results graphically.

These features improve the user’s productivity. Matlab is considerably simpler to work with than programming code segments in Fortran, C, C++, or Java and calling up some visualization program.

As a programming language, Matlab is convenient, but somewhat primitive. The code is interpreted and runs slowly, unless the computations are performed solely in the built-in Fortran and C libraries. This means that one should avoid long loops in Matlab and instead try to perform the computations by combining various Matlab commands whose loops are implemented in Fortran or C (more about this in
Sect. 6.3.7). Recent extensions include the possibility to define classes and to translate Matlab code to C++. Well-trained users will often be able to generate highly efficient Matlab code. For large or advanced scientific computing applications, Matlab can seldom compete with tailor-made programs in the compiled languages (Fortran, C/C++), but for simpler calculations and testing ideas, Matlab is a very popular and productive tool.

The trapezoidal integration method is to be implemented in a Matlab function. Any Matlab function we want to call from the interactive Matlab environment or from a Matlab script must have its source code written in a file with extension .m. Such files are called M-files. In the present example, we create a file Trapezoidal.m, containing more or less a direct translation of Algorithm 6.2:

```matlab
function r = Trapezoidal(a, b, f, n)
% TRAPEZOIDAL Numerical integration from a to b
% by the Trapezoidal rule
f = fcnchk(f);
h = (b-a)/n;
s = 0;
x = a;
for i = 1:n-1
    x = x + h;
    s = s + feval(f,x);
end
s = 0.5*(feval(f,a) + feval(f,b)) + s;
r = h*s;
```

The return parameter is called r and must be assigned before the end of this function. Comment lines start with %. The first comment segment after the heading of the function is taken as a documentation of the function and its usage if the first word is the name of the function. Writing help Trapezoidal in Matlab will then print out this documentation.

In Matlab, functions can be passed as arguments to other functions in various ways. The statement f=fcnchk(f) makes a unified function representation f out of different types of supplied function representations (names of M-files, strings with function expressions, inline Matlab function objects).

The next three statements should be trivial to understand. The semicolon at the end of each statement is not strictly required, but if you omit it, Matlab will echo the statement to the screen. This is inconvenient (and takes much time) if the program executes many statements (which is the case if you have loops).

For loops in Matlab are of the form for variable = expr, where expr is a loop expression. Typical loop expressions are

- a:b, which generates the sequence a, a+1, a+2, ..., b, and
- a:s:b, which generates the sequence a, a+s, a+2*s, ..., b.

In the present example we need an expression with step 1 from 1 to n. To call the function f with argument x, we need to use the Matlab construction feval(f,x).

A function to be integrated is naturally placed in another M-file, say, f1.m:

```matlab
function y = f1(x)
y = exp(-x*x)*log(1+x*sin(x));
```
To integrate \( f_1 \), we can call the \texttt{Trapezoidal} function in the interactive Matlab environment as follows:

\begin{verbatim}
    a = 0; b = 2; n = 10;
    result = Trapezoidal(a, b, @f1, n);
    disp(result);  % print result
\end{verbatim}

Notice that a function \( f_1 \) in an M-file \texttt{f1.m} is transferred as an argument by adding the @ prefix to the name \( f_1 \).

Adding timing functionality is easy with the Matlab \texttt{cputime} function. However, we need to run a long loop over the \texttt{Trapezoidal} function call to obtain some seconds of CPU time:

\begin{verbatim}
    a = 0; b = 2; n = 1000;
    t0 = cputime;
    for i = 1:10000  % repetitions to obtain some seconds CPU time
        result = Trapezoidal(a, b, @f1, n);
    end
    disp(result);
    t1 = cputime - t0;
    disp(t1);
    exit
\end{verbatim}

Matlab allows a flexible assignment of functions, as demonstrated next:

\begin{verbatim}
    a = 0; b = 2; n = 10;
    % function \( f_1 \) defined in f1.m (function handle):
    result = Trapezoidal(a, b, @f1, n);
    disp(result);
    % inline object \( f \):
    f = inline('exp(-x*x)*log(1+x*sin(x))');
    result = Trapezoidal(a, b, f, n);
    disp(result);
    % string expression:
    result = Trapezoidal(a, b, 'exp(-x*x)*log(1+x*sin(x))', n);
    disp(result);
\end{verbatim}

Running the simple main program in Matlab on a laptop resulted in a CPU time ratio of 85 relative to the Fortran 77 and C/C++ codes. This can, however, be dramatically improved by vectorizing the code, see Sect. 6.3.7.

### 6.3.6 Python

Python is a very flexible and convenient programming language that supports much more advanced concepts than C or Fortran, and also more powerful constructions than in C++ or Java. The nature of Python allows one to build libraries with an interface that gives the programmer access to powerful high-level statements. Application codes therefore tend to be smaller, more compact, and easier to read than their counterparts in Fortran, C, C++, and Java. Because Python programs are interpreted, some constructions (especially loops) run much more slowly than in
Fortran, C, C++, or Java. However, for many applications Python is fast enough, in scientific computing contexts as well [19]. Only a glimpse of Python is given below. The reader is referred to the book [20] for a more comprehensive treatment of Python’s applicability in a numerical context.

Matlab and Python have much in common: Both are easy to learn, they have a very clean syntax, and they have high-level tools for performing compound operations in a few statements. Matlab has more built-in functionality for scientific computing, but Python is a more advanced and flexible programming environment. A good strategy is to use Matlab for tasks where Matlab is strong, mainly linear algebra-related problems, and build your own, tailored Matlab-like environment via Python for more advanced tasks. Of course, Python can also be used for linear algebra problems.

Variables in Python can readily be brought into play, without explicitly mentioning their type. Since the Python syntax (like Matlab’s) is close to mathematical notation, the difference between Algorithm 6.2 and Python code is small:

```python
#!/usr/bin/env python
from math import *

def Trapezoidal(a, b, f, n):
    h = (b-a)/float(n)
    s = 0
    x = a
    for i in range(1,n,1):
        x = x + h
        s = s + f(x)
    s = 0.5*(f(a) + f(b)) + s
    return h*s

def f1(x):
    f = exp(-x*x)*log(1+x*sin(x))
    return f

def f2(x): # simple function for verification
    return 1 + x # integral from 0 to 2 should be 4

a = 0; b = 2; n = 1000
for i in range(10000):
    result = Trapezoidal(a, b, f1, n)
import time # measure time spent in the program
    t1 = time.clock() # CPU time so far in the program
    print result, t1
```

This is the complete code; the trapezoidal function, the test function f1 to be integrated, and a main program calling up the integration 1,000 times such that measurements of the CPU time become reliable. As in most other languages, we need to import library functionality, here through the import statement. Most of the syntax should be self-explanatory, even for the novice with the exception, perhaps, that functions are prefixed by def and for loops from p to q in steps of r are written for

---

8 The intuitive treatment of functions as arguments in functions, cf. the f variable in function Trapezoidal, is considerably simpler and clearer than the corresponding treatment of function arguments in the other languages we treat herein.
i in range(p, q+1, r), or just for i in range(q) if the indices from 0 up to and including q-1 are wanted (these for loops can only work with integer counters).

Python codes are interpreted. Hence, there is no need to compile the file containing the code; just write the name of the file, `int.py`, to execute it under Unix, or write

```python
python int.py
```

which works under all operating systems where Python is installed.

Python also allows interactive computing. You can write `python` on the command line to enter Python’s interactive mode, but we recommend using the more user-friendly IDLE shell that comes with the Python source code distribution. Start with

```python
from int import *
```

where `int.py` is the complete Python program shown above. The `import` statement causes all the statements in `int.py` to be executed, including the integral computations at the end of the file. The nice thing is that we have, through the `import` statement, defined the functions `Trapezoidal` and `f1`, so we can easily experiment with, e.g., the `n` value as follows:

```plaintext
>>> from int import *
0.25046779824770.6
>>> n=10; Trapezoidal(a, b, f1, n)
0.25021765112667527
>>> n=1000; Trapezoidal(a, b, f1, n)
0.25046779824710441
>>> n=3; Trapezoidal(a, b, f1, n)
0.24752392127618572
```

The Python program executed on a laptop machine resulted in a CPU time ratio of 14 relative to the Fortran 77 and C/C++ codes. The next section explains how to significantly improve the CPU time.

### 6.3.7 Vectorization

We have seen that both Matlab and Python enable implementation of the trapezoidal rule close to the syntax used in Algorithm 6.2, but the code is interpreted and runs much more slowly than in Fortran, C, C++, and Java. Loops in interpreted languages will run slowly; actually, such languages are not meant for intensive numerical computations in explicit for loops – this is a kind of misuse of the

---

9 If `PYTHONSRC` is the root of the Python source code tree, the IDLE shell is launched by executing `PYTHONSRC/Tools/idle/idle.py`. You should make an alias for this (long) path in your working environment.

10 The CPU time of the computations is significant, so you would probably get rid of the repetitive calls to `Trapezoidal` before using `int.py` in interactive mode.
language. Instead, interpreted languages often offer alternative programming techniques for achieving efficiency in computations with large data structures or long loops.

Basic Ideas of Vectorization

Algorithms with long for loops in Matlab and Python should be vectorized. Vectorization means expressing mathematical operations in terms of vector operations instead of loops with element-by-element computations. A simple example may be summing up the square of the elements of a vector \((v_1, \ldots, v_N)\):

\[
s = 0 \\
\text{for } i = 0, \ldots, N \\
\quad s \leftarrow s + v_i^2 \\
\text{end for}
\]

In Matlab this translates to

```matlab
s = 0; 
for i = 1:N 
    s = s + v(i)*v(i); 
end
```

Similar Python code reads

```python
s = 0 
for i in range(N): 
    s = s + v[i]*v[i]
```

These loops normally run slowly. To vectorize the code, we need to express the loops in terms of vector operations. This is very simple in the present case, because \(s\) is nothing but the square of the norm of \(v\).

Matlab has a function \texttt{norm} that we can use:

```matlab
s = norm(v); s = s*s
```

Python has a special package, Numerical Python, for efficient array storage and computation. This module does not provide a norm function, but the norm of \(v\) is the square root of the inner product (also called the scalar product) of \(v\) and itself. Since Numerical Python offers an \texttt{innerproduct} function, we can compute \(s\) as

\[
s = \text{innerproduct}(v,v)
\]

The major difference between computing \(s\) through a for loop and through a \texttt{norm} or \texttt{innerproduct} function is that the latter functions are implemented as a for loop in C or Fortran. Therefore, \texttt{norm} in Matlab and \texttt{innerproduct} in Python run as fast as highly optimized C or Fortran code.

The idea of vectorization is to avoid explicit loops and instead invoke C or Fortran functions where the loop can be much more efficiently implemented. However, vectorizing an algorithm can require quite some rewriting of a loop-based algorithm. The rewriting also depends on what vector operations are offered by the
programming language. The linear algebra field in mathematics defines some vector operations, such as inner (scalar) products, matrix-vector products and so on. Programming languages support these mathematical operations, but also offer additional functionality. Unfortunately, there is no standard naming convention for this additional functionality. The vectorization of an algorithm is therefore more tightly coupled to its actual implementation in a particular programming language.

The Trapezoidal Rule in Vectorized Form

Vectorization of Algorithm 6.2 requires rewriting the instructions such that the loop is replaced by built-in operations working on vectors. Let us start with a vector \( x \), going from \( a \) to \( b \) in steps of \( h = (a - b)/n \):

\[
x_1 = a, \quad x_2 = a + h, \ldots \quad x_j = a + (j - 1)h, \ldots \quad x_{n+1} = b.
\]

We then apply \( f \) to each element in \( x \) such that \( f(x) \) is a new vector \( v \) of the same length as \( x \). This application of \( f \) to \( x \) should be performed in Fortran, C, or C++ for efficiency. Languages with support for vector operations normally supply a function for summing all the elements of the vector \( f(x) \): \( s = \sum_{i=1}^{n+1} f_i \). This operation is often named \texttt{reduce} or \texttt{sum}. The quantity \( s \) is not exactly what we want, since the end points \( f_1 \) and \( f_{n+1} \) should have weight \( \frac{1}{2} \) and not 1, as implied in the sum over all elements \( f_i \). The correction is easily performed by subtracting \( \frac{1}{2}(f_1 + f_{n+1}) \). Finally, we need to multiply \( s \) by \( h \). Algorithm 6.13 summarizes the various steps (\texttt{sum(v)} here is our notation for adding all elements in \( v \)).

**Algorithm 6.13**

**Vectorized Trapezoidal Integration.**

\[
\text{Trapezoidal}_\text{vec} (a, b, f, n)
\]

\[
\begin{align*}
\hat{h} &= \frac{b-a}{n} \\
\mathbf{x} &= (a, a + h, \ldots, b) \\
v &= f(\mathbf{x}) \\
s &= \hat{h} \cdot (\text{\texttt{sum}(v)} - 0.5(v_1 + v_{n+1})) \\
\text{return } s
\end{align*}
\]

Vectorized Numerical Integration in Python

Translating Algorithm 6.13 into Numerical Python is straightforward. Generation of the sequence of evaluation points \((a, a + h, \ldots, b)\) is performed by the statement

\[
x = \text{arrayrange}(a, b, h, \text{Float})
\]

Because of round-off errors the upper limit can end up as \( b-h \) instead of \( b \). It is therefore smart to set the upper limit as \( b+h/2 \); this will guarantee that \( b \) becomes
the last element in \( x \). The sum operation on a vector \( v \) is called \( \text{add.reduce}(v) \) in Numerical Python. Knowing about \( \text{arrayrange} \) and \( \text{add.reduce} \), it should be quite easy to understand the implementation of the vectorized algorithm in Python:

```python
def trapezoidal(a, b, f, n):
    h = (b-a)/float(n)
    x = arrayrange(a, b+h/2, h, Float)
    v = f(x)
    r = h*(sum(v) - 0.5*(v[0] + v[-1]))
    return r
```

Indexing Python arrays is a slow process, but we do it only twice, which is not critical if \( n \) is large. Arrays in Python start with index 0, and \(-1\) is a notation for the last index.

Numerical Python redefines the multiplication operator * such that \( h*f(x) \) is a scalar \( h \) multiplied by a vector \( f(x) \). The expression \( f(x) \) means applying a function \( f \) to a vector \( x \), element by element. However, we must avoid implementing \( f(x) \) as an explicit loop in Python. To this end, Numerical Python supports the efficient application of standard mathematical functions and operators on vectors. For example, \( \sin(x) \) computes the sine of each element in a vector \( x \). We can therefore implement a function \( f(x) = e^{-x^2} \ln(1 + x \sin x) \) applied to each element of a vector \( x \) as

```python
def f1(x):
    f = exp(-x*x)*log(1+x*sin(x))
    return f
```

This is exactly the same syntax as that used for applying a scalar function on a scalar value. Sending, e.g., a number 3.1 to \( f1 \), results in the computation \( e^{-3.1^2} \ln(1 + 3.1 \sin 3.1) \). Sending a vector \( x \) to \( f1 \) results in more complicated computations “behind the curtain”:

1. temp1 = sin(x), i.e., apply the sine function to each entry in \( x \)
2. temp2 = 1 + temp1, i.e., add 1 to each element in temp1
3. temp3 = log(temp2), i.e., compute the natural logarithm of all elements in temp2
4. temp4 = x*x, i.e., square each element in \( x \)
5. temp5 = exp(temp4), i.e., apply the exponential function to each element in temp4
6. \( f = \text{temp5*temp3} \), i.e., multiply temp5 and temp3 element by element, as in a scalar multiplication

As we see, \( f \) is built of several vector operations, requiring temporary arrays, and hence additional storage, compared with a straightforward loop. However, a straight loop computing \( f \) element by element required 40 times as long CPU time as the vectorized version (on an Intel laptop computer with Linux and Python compiled by GNU’s C compiler \( gcc \)).

The main program calling up the vectorized version of the \( \text{trapezoidal} \) function is exactly the same program as we used to test the scalar version of \( \text{trapezoidal} \):
a = 0; b = 2; n = 1000
for i in range(10000):
    result = trapezoidal(a, b, f1, n)
import time
t1 = time.clock()
print result, t1

We note that in Python you can say `print a`, where `a` is any built-in object (including an array) and get nicely formatted output. This is very convenient during program development and debugging.

The reader should notice how convenient a language using dynamic typing is; many of the code segments developed for scalar computations can be directly reused for vectorized computations. (The same functionality is obtained in C++ by the use of templates and overloaded operators, though at the cost of increased language and technical complexity.)

The vectorized version of the trapezoidal method uses one-seventh of the CPU time required by the loop-based counterpart. This means that vectorized Python runs slightly faster than Java and at about half the speed of Fortran 77, C, or C++.

Vectorized Numerical Integration in Matlab

Algorithm 6.13 can be implemented directly in Matlab:

```matlab
function r = Trapezoidal_vec(a, b, f, n)
    % TRAPEZOIDAL Numerical integration from a to b
    % by the Trapezoidal rule
    f = fcnchk(f);
    h = (b-a)/n;
    x = [a:h:b];
    v = feval(f, x);
    r = h*(sum(v) - 0.5*(v(1) + v(length(v))));
```

The construction of the sequence of evaluation points having a uniform distance `h` between `a` and `b` is created by the expression `[a:h:b]`. The `feval(f,x)` expression we know from the scalar (non-vectorized) version of the function can also be used when `x` is a vector. As in Python, the return value from `f` is then a vector. Summing up all elements in a vector `v` is performed with `sum(v)`, and indexing the last element in `v` can be written as `v(length(v))` (if the first index in `v` is set to 1, which is the default value).

The function to be integrated must now work for a vector. In the previous Python example we could reuse a function implemented for a scalar input and return value. Such reuse is not possible in Matlab when the function expression `f(x) = e^{-x^2} \log(1 + x \sin x)`. The reason is because the expression `exp(-x*x)` for a vector `x` implies a matrix multiplication between `x` and itself. What we need is the `.*` operator such that an element in the answer is the product of the corresponding two elements in the operands. Hence, the vectorized version of the scalar `f1` function is

```matlab
function y = f1_vec(x)
    y = exp(-x .* x) .* log(1 + x .* sin(x));
```
We store this function in an M-file `f1_vec.m`. The main program used to test the efficiency of the scalar version of the trapezoidal rule implementation in Matlab can be reused in the vectorized case if the name of the integration routine and the function to be integrated are changed:

```matlab
function y = f1_vec(x)
    y = exp(-x .* x) .* log(1 + x .* sin(x));
```

The vectorized version above reduced the CPU time to 1/42nd of the CPU time required by the loop-based Matlab scripts from Sect. 6.3.5. The CPU times of the vectorized versions of the Python and Matlab codes were approximately equal, running at half the speed of the Fortran 77 and C/C++ implementations.

Discussion

It might seem complicated to use languages that require vectorization for computational efficiency. Straightforward implementations in terms of loops are indeed simpler, but vectorization has one great advantage: The vector operations can utilize a library particularly tuned for the hardware in question. For example, smart tricks to speed up array operations can be implemented in the library, but more importantly, the library can implement parallel versions of the array operations such that one can utilize parallel computers with only minor competence in parallel programming. At least in principle, it may be hard to write a straightforward loop that can compete with highly tuned basic array operations in libraries. The present example is sufficiently simple such that straightforward loops inside a library yield high (and close to optimal) efficiency.

Vectorization became an important programming technique with the advent of the first supercomputers in the late 1970s. We just argued that in a compiled language like Fortran there is no need to transform loops into calls to vector operations. The situation was a bit different for the first supercomputers. The point then was to break up more complicated nested loops into a series of simpler loops performing basic vector operations. These simpler loops could easier be translated by the compiler to a suitable form for very efficient computations on the supercomputers. This type of vectorization can still be beneficial on modern RISC architectures found in PCs and workstations. More importantly, basic vector operations have been implemented with maximum efficiency, tuned to the hardware, and made available in libraries (BLAS and ATLAS are examples). To get the ultimate speed out of code, it may be advantageous to rewrite the algorithms and implementations such that they can utilize very efficient libraries for vector operations.

In the 1980s, supercomputers were based on parallel computations, and algorithms had to be recast again, this time from serial into parallel versions. The importance of vectorization disappeared, but due to the popularity of interpreted environments such as Matlab and Python, RISC-based computers, and standardized libraries for vector operations, the vectorized programming style remains highly relevant.
6.3.8 Maple

Maple is a complete problem-solving environment for symbolic calculations. Symbolic in this context means manipulating mathematical expressions, not just numbers. For example, Maple can calculate the integral of \( \cos x \) to be \( \sin x \) and differentiate \( \sinh x \) to get \( \cosh x \). Maple is also equipped with the widely used NAG libraries for efficient numerical computations.

A particularly attractive feature of Maple is that you can write interactive scientific reports with formulas, text, and plots. The report can be read as is, but the reader can also change Maple commands (formulas, equations, etc.) and view the consequences of such changes. This encourages explorative and interactive investigations of scientific problems.

Maple is a complete programming language. As in Matlab and Python, variables are not explicitly typed and there are many high-level commands, making Maple programs compact.

The bridge between Algorithm 6.2 and a Maple function is short:

```maple
Trapezoidal := proc(a,b,f,n)
local h, s, x, i:
h := (b-a)/n:
s := 0: x := a:
for i from 1 to n-1 do
  x := x + h:
s := s + f(x):
end:
s := s + 0.5*f(a) + 0.5*f(b):
s := h*s:
s;
end:
```

Maple performs symbolic computations. That is, \((b-a)/n\) is not a number unless \(a\), \(b\), and \(n\) are assigned numbers; if these arguments are assigned mathematical expressions, \((b-a)/n\) remains an expression, i.e., a mathematical formula instead of a number. In the present case, \(a\), \(b\), and \(n\) are numbers, so the operations in the trapezoidal function are numerical computations.

A test function \(f1\) to be integrated is defined by

```maple
f1 := x -> exp(-x*x)*log(1+x*sin(x));
```

The numerical integration of \(f1\) is now carried out by

```maple
q := Trapezoidal(0, 2, f1, 1000)
```

This code runs very slowly. Repeating it 10 times and multiplying the CPU time by 1,000 (corresponding to 10,000 repetitions as we have used for all previous CPU time comparisons) showed that Maple required more than 600 times the CPU time of the Fortran 77 implementation! We should add that the speed of Maple varied greatly (up to a factor of 4) between different executions.

The reason why the implementation of Algorithm 6.2 runs so slowly in Maple is that Maple code is interpreted. Long for loops are, in some sense, a misuse of Maple. One should either implement such loops with numerical computations in
C or Fortran code to be called from Maple, or (better) use a built-in Maple function to perform the task. In the case of numerical integration, Maple offers the user many computationally efficient functions. The plain trapezoidal rule is not available. Maple instead offers sophisticated methods with user interfaces for specifying the desired precision and a particular rule rather than the detailed parameters of a method. Maple then calls up highly optimized Fortran 77 functions, which automatically find a rule compatible with the user’s accuracy requirement.

In the present case, we just write

```maple
q := evalf(Int(f1, 0..2, 'method'=_NCrule));
```

to evaluate the integral of \( f1 \) numerically by the simplest rules in Maple. The answer is returned in a fraction of a second. Maple computes our integral faster than the Fortran 77 program and with higher precision. The reason is because Maple applies a much more efficient numerical integration rule.

To summarize, we can easily implement Algorithm 6.2 in Maple, but the execution time will be long if \( n \) is large. The “right” solution is to use the built-in numerical integration functions, but this invokes much more sophisticated and efficient methods than Algorithm 6.2.

### 6.3.9 Summary

In the previous sections we have presented implementations of the trapezoidal rule in different computing environments. Although the trapezoidal rule is among the very simplest of all numerical algorithms, quite significant differences in performance and convenience of the various programming tools have come to be. These differences are much larger and more serious when addressing more complicated numerical algorithms.

It is almost impossible to draw firm conclusions about the preferred computing environment, since this depends heavily on personal taste and previous experience. We have tried to define some characteristics of the different programming tools in Table 6.1 when implementing the trapezoidal rule. The purpose is to provide a quick comparison of some features that programmers of numerical applications may emphasize when choosing a language.

Some trends are clear when comparing Fortran 77, C/C++, Java, Python, Matlab, and Maple. If computational speed is the most important factor, one should choose one of the compiled languages: Fortran 77, C, C++, or Java. If programming convenience is regarded as most important, the interpreted and dynamically typed environments Maple, Matlab, and Python constitute the preferred choice. Speed versus convenience can also be expressed as computational versus human efficiency. The former is very important if the total CPU time is hours, days, or weeks, or if the program is to be executed “as is” in thousands of production runs by many people. Students or scientists will often be most concerned with human
efficiency. Some programmers regard C++ as a “middle of the road” choice: It provides speed and a fair amount of programming convenience.

Maple is an obvious choice if it is important to integrate symbolic and numerical computing. However, Matlab has a communication link to Maple, and Python can call up symbolic mathematics engines such as SymPy or Ginac, so integration of symbolic functionality can be performed in all three environments.

Even if a particular feature is supported in several languages, the usage of the feature may be considerably more favorable in one particular language. Classes constitute an example; the authors find class programming easiest and most convenient in Python, followed by Java and C++, while we find Matlab classes less convenient. These finer points are not easy to express in a tabular set-up such as Table 6.1.

The most obvious conclusion is that there is no single preferred scientific computing tool. You need to gain experience with several and learn how to pick the most productive tool for a given task. We recommend you gain this knowledge from “learning by doing”, so that you can build up your own experience rather than trying to deduce the conclusion from theoretical considerations.

Finally, we mention that only some relevant programming languages are compared here. We have said little about C and not mentioned Python’s “brothers” Perl and Ruby; there are also excellent computing environments such as Mathematica, Gauss, and R (or S-Plus) that are not mentioned here. We have also neglected the modern Fortran versions: 90, 95, and 2000. These represent a smooth transition from the simple Fortran 77 language to a rich language such as C++. For many purposes, Fortran 2003\footnote{At the time of this writing, Fortran 2003 compiler are just starting to emerge.} shares the same characteristics as C++, with some convenient additional built-in features for numerical computing.

**Table 6.1** Comparison of different programming languages. The numbers given in the “speed” column equal the actual CPU time of Algorithm 6.2, implemented in the particular language, divided by the CPU time required by the associated Fortran 77 implementation. For Python and Matlab we have reported the CPU time of the vectorized version of Algorithm 6.2. The numbers in the “program length” column correspond to the number of lines in our implementation, including comment lines, and depend significantly on the personal tastes and habits of the programmer. The dashes indicate that it is difficult to give a precise yes or no answer: Maple’s speed is improved by calling built-in numerical functionality, but vectorization as a programming technique is not required; Java and Python (and C++ to some extent) can easily incorporate visualization in programs, but this is not a part of standard libraries.

<table>
<thead>
<tr>
<th>Language</th>
<th>Program length</th>
<th>Speed</th>
<th>Vectorization required</th>
<th>Dynamic typing</th>
<th>Classes</th>
<th>Interactive computing</th>
<th>Built-in visualization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran 77</td>
<td>41</td>
<td>1.0</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>C++</td>
<td>38</td>
<td>1.0</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Java</td>
<td>50</td>
<td>2.1</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>–</td>
</tr>
<tr>
<td>Python</td>
<td>24</td>
<td>1.9</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Matlab</td>
<td>21</td>
<td>2.0</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Maple</td>
<td>36</td>
<td>613.6</td>
<td>–</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
6.4 Heun’s Scheme in Different Languages

The purpose of this section is to implement Heun’s method from Algorithm 6.12 on page 212 in the following programming languages: Fortran 77, C, C++, Java, Matlab, Maple, and Python. Section 6.3 gives a glimpse of these languages applied to the trapezoidal integration technique. You should therefore be familiar with Sect. 6.3 before proceeding with the implementation examples regarding Heun’s method. These new examples involve array and file handling in the various computer languages.

6.4.1 Code Structure

Before diving into the details of various programming language, we should have a clear view of the structure of the complete program. It would be convenient to have $\Delta t$ and the total simulation time $t_{stop} = N\Delta t$ as input parameters to the main program. Algorithm 6.12 must then be called and the solution should be displayed graphically. The latter task can in some environments easily be done inside the program, while in other situations we need to store the solution in a file and use a separate plotting program to view the graph of $u(t)$. We have decided to always write the solution to a file (to demonstrate file writing) and in addition display the solution directly when this is feasible. The code must also implement a right-hand side function $f(u)$. For testing purposes, we choose $f(u) = -u$ and $u(0) = 1$ such that the exact solution reads $u(t) = e^{-t}$.

6.4.2 Fortran 77

Algorithm 6.12 can be directly translated to Fortran 77:

```fortran
subroutine heun (f, u0, dt, n, u)
    integer n
    real*8 u0, u(0:n), dt, f
    external f

    integer i
    real v

    C initial condition:
    u(0) = u0

    C advance n steps:
    do i = 0, n-1
        v = f(u(i))
        u(i+1) = u(i) + 0.5*dt*(v + f( u(i) + dt*v ))
    end do

    return
end
```
There are two new issues (compared with the trapezoidal function explained in Sect. 6.3.2):

1. We now have a subroutine, i.e., a subprogram that does not return a value.
2. One of the arguments is an array. The indices in this array range from 0 up to (and including) \( n \), specified by the declaration \( \text{real*8 } u(0:n) \). The array is indexed\(^{12}\) using the syntax \( u(i) \).

All arguments in Fortran 77 subroutines and functions are both input and output variables, in the sense that any changes in these variables are visible outside the function. That is, if we change \( u \), which we intend to do, since \( u \) is supposed to hold the computed function values, the final \( u \) is immediately available in the calling program:

```fortran
C calling code:
integer n
real*8 v(0:n), v0, some_f, dt
external some_f

dt = 0.01
v0 = 1.0
call heun (some_f, v0, dt, n, v)
```

Note that subroutines are called using the `call` statement (in most other languages it is sufficient to write the name of the subprogram). After the return from the `call heun` statement, the \( v \) array will contain the computed discrete \( u(t) \) values.

If we accidentally change \( u_0 \) inside the `heun` subroutine, the new value will be reflected in the \( v_0 \) variable in the calling program. This is not a feature we want. Most other programming languages have mechanisms for taking a copy of the arguments (usually referred to as call-by-value). Using copies of pure input variables reduces the danger of unintended changes to the variables in the calling code.

Having called the subroutine `heun` to compute discrete function values in an array, we may want to dump this array to file for later plotting. A suitable file format is to have two numbers on each line, the time point and the corresponding value of \( u \), e.g.,

<table>
<thead>
<tr>
<th>Time</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.004</td>
<td>0.996008</td>
</tr>
<tr>
<td>0.008</td>
<td>0.992031936</td>
</tr>
<tr>
<td>0.012</td>
<td>0.988071745</td>
</tr>
<tr>
<td>0.016</td>
<td>0.984127362</td>
</tr>
<tr>
<td>0.020</td>
<td>0.980198726</td>
</tr>
<tr>
<td>0.024</td>
<td>0.976285772</td>
</tr>
<tr>
<td>0.028</td>
<td>0.97238844</td>
</tr>
</tbody>
</table>

Most plotting programs can read such data pairs: \( t_i, u(t_i) \), and draw straight line segments between the points to visualize the curve. In our code we include a subroutine `dump` that creates such a file. Some key statements for illustrating writing in Fortran 77 are given next.

\(^{12}\) Some compilers have flags for checking the validity of the index (with an associated efficiency penalty).
integer iunit
real*8 v
...
iunit = 22
open(iunit, name='sol.dat', status='unknown')
write(iunit, *) 'some text and variables', v, ' and text'
...
close(iunit)

Note that a file is referenced by an integer, iunit here. Our dump subroutine assumes that a file has already been opened, such that we can pass iunit as an input argument and work with it inside the subroutine:

subroutine dump (iunit, u, n, t0, dt)
integer n, iunit
real*8 u(0:n), t0, dt

integer i
real*8 time

time = t0

do i = 0, n
   write(iunit,*) time, u(i)
   time = time + dt
end do
return
end

The main program is listed next and commented upon afterward:

program ode
integer nmax, n, i, iunit
parameter (nmax=1000)
real*8 dt, tstart, tstop, u0, u(0:nmax)
external f1

  tstart = 0.0
write(*,*) 'stop time:'
read(*,*) tstop
write(*,*) 'time step:'
read(*,*) dt

C set a reasonable time step if dt<0:
if (dt .lt. 0.0) then
   dt = 0.004
end if

C check that the u array is sufficiently large:
n = tstop/dt
if (n .gt. nmax) then
   write(*,*) 'ERROR: too small time step'
   end if

C time integration:
u0 = 1.0
call heun (f1, u0, dt, n, u)
write(*,*) 'end value =', u(n), ' error = ', u(n)-exp(-dt*n)

C write solution to file (for plotting):
iunit = 21
open(iunit, name='sol.dat', status='unknown')
call dump (iunit, u, n, tstart, dt)
close (iunit)
end
Arrays in Fortran 77 must be allocated at compile time. That is, we cannot ask
the user for $\Delta t$ and $t_{\text{stop}}$, compute $N = t_{\text{stop}}/\Delta t$, and create an array $u$ of length $N$. Instead, we must allocate a sufficiently long $u$ array in the main program. All
subroutines or functions can, however, receive arrays with lengths specified by variables. In our case, subroutine $\text{heun}$ works with the first $n+1$ elements of the $u$ array
allocated at compile time in the main program. The parameter statement enables
setting the value of constants at compile time.

To get user information about $\Delta t$ and $t_{\text{stop}},$ we use write and read statements
and communicate with the user in terms of questions and answers. In other program-
ing languages we will get input from the command line instead. This is normally
possible in Fortran by calling up functionality in a C library, but the feature is not
a part of the Fortran standard. We also mention that comparisons of numbers in if-
statements apply the operators $\lt,$ $\le,$ $\equiv,$ $\ge,$ and $\gt$ rather than $<$, $\le$, $=$
or $=,$ $\ge,$ and $>$, respectively.

Checking that $n$ is not greater than $n_{\text{max}}$ is very important. Without this test,
subroutine $\text{heun}$ will index the $u$ array outside the allocated chunk of numbers.
This may or may not produce strange error messages (usually just segmentation
fault) that are hard to track down. Some compilers have flags for detecting if the
array index is outside valid values. However, in subroutine $\text{heun}$ $u$ is declared as
$u(0:n)$, and the compiler-generated check is just that the index $i$ does not exceed
the value $n$, not whether $i$ is outside the physical bounds of the array.

The Fortran 77 code is compiled, linked, and run as explained in Sect. 6.3.2.

### 6.4.3 Graphics

Here we explain how we can plot curve data stored in files. The data consists of
points $(x_i, y_i), \ i = 0, \ldots, N$, and the goal is to draw line segments between these
points in an $xy$ coordinate system. We assume that the $(x_i, y_i)$ points are stored
in two columns in a file, the $x_i$ values in column 1 and the $y_i$ values in column 2.
This file is given the name sol.dat in the forthcoming demonstrations of plotting
programs. In the present example with the numerical solution of ODEs, the $x_i$ values
correspond to points in time, and the $y_i$ values are the associated values of the
unknown function $u$ in the differential equation.

**Gnuplot**

A widely available plotting program is Gnuplot. Assuming that the name of the file
containing the data points in two columns is sol.dat, the Gnuplot command

```
plot 'sol.dat' with lines
```

displays the graph. To try it out: Start Gnuplot by typing gnuplot. This enters a
command mode where you can issue the command above.
Normally, we want to set a heading in the plot and also produce a hardcopy of the plot in PostScript format (for inclusion in reports). The following set of Gnuplot commands perform these tasks:

```plaintext
set title "Heun’s method for du/dt = -u"
# plot on the screen:
plot 'sol.dat' title 'u(t)' with lines
# hardcopy in PostScript format:
set term postscript eps monochrome dashed 'Times-Roman' 20
set output 'tmp.ps'
plot 'sol.dat' title 'u(t)' with lines
```

We recommend putting such commands in a file, say `plot`, and executing Gnuplot with the name of this file as the argument:

```
unix> gnuplot -persist plot
```

The `-persist` option makes the plot stay on the screen after the Gnuplot program has terminated. The PostScript version of the plot is displayed in Fig. 6.1.

**Matlab**

Having the $x_i$ values as a vector $x$ and the $y_i$ values as a vector $y$, plotting in Matlab is easily done through the standard command `plot(x,y)`. In our example with $u(t)$ data, the following Matlab commands create a plot of the data in the file `sol.dat` and annotate a title and curve label, in addition to making a hardcopy of the plot:

```matlab
data = load('sol.dat'); % load file into two-dim. array data
t = data(:,1); u = data(:,2); % extract columns as vectors
plot(t, u);
```
Heun’s method for $\frac{du}{dt} = -u$

**Fig. 6.2** Plot made by Matlab

```matlab
    title('Heun method for du/dt = -u');
    legend('u(t)'); % attach curve label
    print('-deps', 'tmp.eps'); % make Encapsulated PostScript plot
```

Suppose these commands are stored in a file `plotmatlab.m`. Inside Matlab we can then write `plotmatlab` to execute the commands in `plotmatlab.m` and thereby create the plot. Alternatively,

```
unix> matlab -nodesktop -r plotmatlab
```

creates the plot and invokes Matlab in a terminal window. Type `exit` to terminate the session. The PostScript version of the plot, as created by Matlab, is displayed in Fig. 6.2.

### 6.4.4 C++

**The Computational Algorithm**

The implementation of Algorithm 6.12 in the C++ programming language can take the following form:

```c++
    void heun (fpTR f, double u0, double dt, int n, double u[])
    {
        u[0] = u0; // initial condition
        // advance n steps:
        for (int i = 0; i <= n-1; i++) {
            double v = f(u[i]);
            u[i+1] = u[i] + 0.5*dt*(v + f( u[i] + dt*v ));
        }
    }
```
The heun function is like a subroutine in Fortran, i.e., it does not return any value, which in C++ is indicated by declaring the return type as void. The f argument is used to transfer the right-hand side of the ODE to this function (see page 222 for details). For our test program, the following f1 function will be supplied as the f argument to the heun function:

\[
double f1 (double u) \{ \text{return } -u; \}
\]

Plain Arrays in C++

The argument u in the heun function is a basic C++ (or C) one-dimensional array, with one index starting at 0. The array argument is declared as double u[] or double* u. The latter notation actually declares a pointer to a double, i.e., the memory address of a double. Arrays and pointers in C and C++ are very closely related; array variables are represented by pointers to (i.e. the address of) the first array element in memory. Indexing arrays is done with square brackets: u[0], u[1], and so on. The array length (here n+1) must be transferred as a separate argument, as in Fortran 77. Another similarity with Fortran is that there is no built-in check of the validity of the array indices. Indexing arrays beyond their bounds is one of the most common errors in these programming languages.

Arrays in C and C++ can be created at run time with an appropriate length after some input data are read. The syntax is as follows:

\[
\text{double* u = new double[n+1];}
\]

The new operator allocates a chunk of memory of sufficient size for storing n+1 double-precision real numbers and returns a pointer to the first array element. We can assign the returned pointer to a pointer variable u and index the array as u[i]. When there is no further use for the array, it must be deallocated (deleted from memory) by the syntax

\[
delete \[u;\]
\]

For every new there should a corresponding delete action. Issuing a delete twice is a common error, and the result is just strange program behavior and abort messages such as segmentation fault. Matching all the new and delete statements correctly appears to be one of the greatest challenges when developing C and C++ code.

Array Objects

Fortunately, C++ has safer array variables. The class valarray in the standard C++ library offers a vector type (array with a single index). Here is a declaration of an array u of length n+1, where each array element is a double:

\[
\text{13 Recall that Fortran arrays are declared with a fixed size at compile time.}
\]
std::valarray<double> u(n+1);

The \( u \) array is automatically deleted when it is no longer in use. The declaration of arrays of type `valarray` in a function such as `heun` is as follows:

```cpp
void heun (fptr f, double u0, double dt, int n,
           std::valarray<double>& u)
```

The ampersand `&` indicates that a reference to the \( u \) object is transferred; without the ampersand, \( u \) will be a copy of the transferred array. This is not what we want, since our goal is to compute values in \( u \) and view these values outside the `heun` function. The syntax of the body of the `heun` function is independent of whether we use `std::valarray` or plain C/C++ arrays.

Multi-dimensional arrays in C and C++ can be constructed, but the allocation and use of `new` is more complicated. Details can be found in books on the C or C++ programming language. Unfortunately, the standard C++ library does not offer a class such as `valarray` for multi-dimensional arrays. One can, nevertheless, create a class for multi-dimensional arrays with minor effort. There are also several open source and commercial libraries in C++ that offer sophisticated multi-dimensional arrays for numerical computing.

File Writing

Basic file writing in C++ is illustrated next:\(^\text{14}\):

```cpp
std::ofstream out("sol.dat"); // open a file sol.dat
out << "some text " << object << " more text\n";
out.close()
```

A `dump` function for writing the \( u(t) \) values to a file can typically be coded as follows:

```cpp
void dump (std::ostream& out, double u[], int n, double t0, double dt)
{
    double time = t0;
    for (int i = 0; i <= n; i++) {
        out << time << " " << u[i] << '\n';
        time += dt;
    }
}
```

Although the file is of type `std::ofstream`, it is common to declare the corresponding function argument as `std::ostream`, because this type is a common name for all types of output media. For example, the `out` argument above can be a file (`std::ofstream`) or the terminal window (`std::cout`), or even a string.

---

\(^\text{14}\)This is C++ file writing. One can also use the C functionality for file writing. This can be convenient if format control via the `fprintf` function is desired.
The Main Program

The main program (in the case where we use a primitive C/C++ array for u) is listed below.

```c
int main (int argc, const char* argv[]) {
    // check that we have enough command-line arguments:
    if (argc < 3) {
        std::cout << "Usage: " << argv[0] << " tstop dt\n";
        exit(1);
    }
    // read tstop and dt from the command line:
    double tstop = atof(argv[1]); // get 1st command-line arg.
    double dt = atof(argv[2]); // get 2nd command-line arg.
    if (dt < 0.0) { dt = 0.004; } // set a suitable time step if dt<0
    int n = int(tstop/dt); // no of time steps
    double* u = new double[n+1]; // create solution array
    // time integration:
    double u0 = 1.0; // initial condition
    heun (f1, u0, dt, n, u); // advance solution n steps
    printf("end value=%g error=%g\n", u[n], u[n]-exp(-dt*n));
    // write solution to file (for plotting):
    std::ofstream out("sol.dat");
    dump (out, u, n, 0.0, dt);
    out.close();
    delete [] u; // free array memory
    return 0; // success of execution
}
```

Command-Line Arguments

In this main program we get input data, such as \( t_{\text{stop}} \) and \( dt \), from the command line. All command-line arguments in C and C++ programs are transferred to the \texttt{main} function as an array of strings. For example, if the name of the executable file is \texttt{ode} and we run

```
unix> ./ode 4.0 0.04
```

there are two command-line arguments: \texttt{4.0} and \texttt{0.04}. These arguments are stored in an array of strings \texttt{argv}. This array has the name of the program (here \texttt{ode}) as its first argument (\texttt{argv[0]}). The first command-line argument is \texttt{argv[1]}, the second is \texttt{argv[2]}, and so on. Observe that all command-line arguments are available as strings, so if we want to compute with \texttt{t_{\text{stop}}}, we need to convert the string to a floating-point number. This is performed by the \texttt{atof} (ASCII to float) function.

The number of array elements in \texttt{argv} is provided as \texttt{argc}. We can thus test if the user has provided a sufficient number of arguments (two in this case). The handling of command-line arguments in C and C++ is reflected in many other languages, including Java and Python.
Graphics

Plotting the solutions is performed as in the Fortran case; i.e., an external plotting program is used to plot the data stored in the file sol.dat.

There exist plotting programs written in C or C++ that can be used as libraries for the present program such that the u array can be sent directly to a plotting routine, without using a file for intermediate data storage. However, there are no “standard” plotting libraries for C++.

6.4.5 Java

The Java program for solving an ODE by Heun’s method is quite similar to the corresponding C++ program, except that the stand-alone functions heun and dump must be methods of a class in Java.

The implementation of the heun function, as described in Algorithm 6.12, is performed via the advance method in class Heun:

```java
class Heun {
    public static double[] advance(Func f, double u0, double dt, int n) {
        double u[] = new double[n+1];
        u[0] = u0; // initial condition

        // advance n steps:
        for (int i = 0; i < n; i++) {
            double v = f.f(u[i]);
            u[i+1] = u[i] + 0.5*dt*(v + f.f(u[i] + dt*v));
        }
        return u;
    }
}
```

The right-hand side of the ODE is represented as function objects in a Func hierarchy, as explained in Sect. 6.3.4. Arrays in Java are created in the same manner as in C++. However, there is no need to delete the array, since Java will do this when the array is no longer in use. This makes it possible to create our array wherever it is convenient, e.g., in the Heun.advance function, and return it from the function to the calling code, if desired. (In C++ this is also possible, but it is usually a good habit, from a programming safety point of view, to perform both the new and delete statements in the calling code. Declaring a variable u of type std::valarray inside the C++ heun function can of course be performed, but C++ requires us to copy this array if we want to return it, whereas with plain C/C++ and Java arrays we avoid this copy, and only references/pointers are returned.)

File writing in Java is typically illustrated by the following statements:

```java
PrintStream f = new PrintStream(new FileOutputStream("sol.dat"));
f.println(time + " " + u[i]);
f.close()
```
Dumping the solution array \( u \) to a file is performed by a `dump` method in a class `IO`:

```java
class IO {
    public static void dump (PrintStream f, double u[], double t0, double dt) {
        int n = u.length;
        double time = t0;
        for (int i = 0; i < n; i++) {
            f.println(time + " " + u[i]);
            time += dt;
        }
    }
}
```

The main program can take this form:

```java
class Demo {
    public static void main (String argv[]) throws IOException {
        double tstop = Double.valueOf(argv[0]).doubleValue();
        double dt = Double.valueOf(argv[1]).doubleValue();
        int n = (int) (tstop/dt);
        Func f = new f1();
        double u0 = 1.0;
        double u[] = Heun.advance (f, u0, dt, n);
        double error = u[n] - Math.exp(-dt*n);
        System.out.println("end value=" + u[n] + " error=" + error);
        PrintStream file = new PrintStream(new FileOutputStream("sol.dat"));
        IO.dump (file, u, 0.0, dt);
        file.close();
    }
}
```

Plotting of the solution is subjected to the same comments as we made for C++:
Calling a Java plotting library is possible, but there are no standard plotting tools to be used.

### 6.4.6 Matlab

In Matlab, we implement Algorithm 6.12 as a function `heun` in an M-file `heum.m`:

```matlab
function u = heun (f, u0, dt, n)
% HEUN numerical integration of ODEs by Heun’s method

f = fcnchk(f);
u = zeros(n+1,1); % make n+1 zeros (as initialization)
% initial condition:
u(1) = u0;
% advance n steps:
for i = 1:n % i <-> i-1 compared to the original algorithm
    v = feval(f, u(i));
    u(i+1) = u(i) + 0.5*dt*(v + feval(f, u(i) + dt*v));
    fprintf(’u(\%d)=%g\n’,i+1,u(i+1))
end
```

As in Java, it is convenient to create arrays wherever we need them and return them from functions, if desired. Matlab will automatically delete arrays when they are no
longer in use. Transfer of arrays in and out of functions is performed very efficiently by transferring references to the arrays.\footnote{This is not an issue for the programmer to worry about. On the other hand, when applying C++, the programmer is actually required to gain the necessary knowledge and control the low-level details of transferring arrays between functions.}

Transferring functions (here the right-hand side, \( f(u) \), in the ODE \( u'(t) = f(u) \)) to the `heun` function follows the set-up from Sect. 6.3.5. We find it convenient to create the \( u \) array inside the `heun` function. The syntax \([0:n] \ast 0\) first creates an array with values 0, 1, up to and including \( n \), and then all the elements in this array are multiplied by zero. There is actually no need to initialize the array elements, since this will be done in the subsequent loop. Arrays in Matlab always start with 1 as the index so we need to rewrite Algorithm 6.12. Although this is easy (just replace \( i \) by \( i + 1 \)), the rewrite is often an error-prone process.

File writing in Matlab is very similar to file writing in C. The basic statements are typically

```matlab
fid = fopen('sol.dat', 'w');
fprintf(fid, 'some text, u[%d]=%g
', i, u(i));
fclose(fid);
```

Here we have used an output function `fprintf`, which is very convenient when we want to control the format of the output. The function follows the syntax of the `printf`-family of functions originating from C. The first argument is a string, where variables can be inserted in “slots” starting with \%. The variables are listed after the format string (here \( i \) and \( u(i) \)). The character after \% determines the formatting of integers, reals, or strings. Here we specify that an integer and a real are to be written as compactly as possible: \%d and \%g, respectively. More sophisticated formatting, e.g., writing a real number in scientific notation with six decimals in a field 12 characters wide can be specified as \%12.6e-4. Complete documentation of the `printf`-syntax is obtained by typing `man sprintf` or `perldoc sprintf` on Unix/Linux systems (in general one can look up the online documentation of Perl’s `sprintf` function).

A `dump` function for writing \( u \) to an already opened file reads

```matlab
function dump (fid, u, t0, dt)
    \% DUMP write array, containing the solution of an ODE, to file
    n = length(u);
    time = t0;
    for i = 1:n
        fprintf(fid, '%g %g
', time, u(i));
        time = time + dt;
    end
```

One should note that strings in Matlab are always enclosed in single quotes.

In the main program there is no natural reading of input data from the command line, since one usually operates Matlab interactively and sets variables directly in the command interpreter. We therefore “hardcode” the input parameters \( t_{stop} \) and \( dt \):
6.4 Heun’s Scheme in Different Languages

% main program for solution of an ODE:
tstop = 2.0;
dt = 0.004;
n = tstop/dt;
u0 = 1.0;
u = heun(@f1, u0, dt, n);
m = length(u);
fprintf('end value=%g error=%g\n', u(m), u(m)-exp(-dt*n));
fid = fopen('sol.dat', 'w');
dump(fid, u, 0.0, dt);
fclose(fid);

Matlab has a wide collection of visualization tools. In particular, it is straightforward to plot Matlab arrays without going through a data file. The commands listed in Sect. 6.4.3 can be issued right after u is computed:

% visualization:
% plot t and u array, i.e., (t(i),u(i)) data points:
t = [0:dt:tstop];
plot(t, u);
title('Heun method for du/dt = -u');
legend('u(t)'); % attach curve label

6.4.7 Python

The Python code for solving the present ODE problem is quite similar to the Matlab code. In general, these two languages usually lead to the same nature of compact, easy-to-understand programs.

Algorithm 6.12 can be implemented directly as a Python function called heun:

def heun (f, u0, dt, n):
   """numerical integration of ODEs by Heun’s method""
   u = zeros(n+1) # solution array
   u[0] = u0 # initial condition
   # advance n steps:
   for i in range(0,n,1):
      v = f(u[i])
      u[i+1] = u[i] + 0.5*dt*(v + f(u[i] + dt*v));
   return u

In Python, we create arrays when we need them. Arrays are efficiently transferred to and from functions using references, and Python automatically deletes arrays that are no longer in use (these two features are shared with Matlab and Java). Python arrays have zero as the base index, as in C++. The programmer can, however, create her own arrays with arbitrary base indices. We emphasize that the arrays we use in Python for numerical computing are created by the Numeric module; the built-in Python lists (also sometimes called arrays) are not well suited for scientific computations.

File writing in Python follows this setup:

file = open("sol.dat", "w") # open file for writing
file.write("some text, u[%d]=%g\n" % (i,u[i]))
file.close()
Here we have used Python’s way of formatting strings with printf-syntax (see page 250 for a brief introduction to printf-style formatting). The Python syntax is a string, containing slots for variables, followed by % and a list of the variables to be inserted in the slots.

The dump function for writing u to a file is now straightforward:

```python
def dump(file, u, t0, dt):
    """write array, containing the solution of an ODE, to file""
    time = t0
    for i in range(len(u)):
        file.write("%g %g\n" % (time, u[i]))
        time += dt
```

The main program, fetching input data from the command line, can be in this form:

```python
if len(sys.argv) <= 2:
    print "Usage: %s tstop dt" % sys.argv[0]
    sys.exit(1)
tstop = float(sys.argv[1])  # 1st command-line arg
dt = float(sys.argv[2])    # 2nd command-line arg
if dt < 0: dt = 0.004
n = int(tstop/dt)

def f1(u): return -u  # right-hand side: du/dt = f(u) = -u
u0 = 1.0
u = heun(f1, u0, dt, n)
print "end value=%g error=%g\n" % (u[-1], u[-1]-exp(-dt*n))
dump(f, u, 0.0, dt)
f.close()
import time; print "CPU-time=", time.clock()
```

The indexing u[-1] means the last element in u, i.e., the same as u[len(u)-1] (u[-2] is the second last element, and so on).

The solution can, of course, be visualized by invoking a plotting program such as Gnuplot or Matlab with the sol.dat file. On the other hand, there is a package called SciTools (developed at the authors’ institution) that offers a Matlab-like set of commands for plotting in Python. Executing

```python
from scitools.std import *
```

imports all of the array and plotting functionality. The relevant Python statements for visualizing the computed u as a curve of (t, u(t)) points then read

```python
# visualize solution:
t = linspace(0, tstop, u.size)
plot(t, u, axis=[t[0], t[-1], 0.1, 1.1], title='u(t)',
     hardcopy='tmp.ps')
```

### 6.4.8 Summary

Section 6.3.9 provides a comparison of Fortran 77, C/C++, Java, Matlab, Python, and Maple for implementing simple functions and loops. The present section adds comments regarding the handling of arrays and graphics. The other new program
feature in the present ODE example, file writing, is handled in a similar way in the various languages.

Array creation and processing is particularly efficient in Fortran. First, arrays are allocated at compile time, with a fixed size (although arrays seemingly have a dynamic size in subroutines and functions). Second, Fortran 77 compilers have been developed over five decades to a very sophisticated level with respect to optimizing (long) loops with array traversal.

Java, Matlab, and Python offer a user-friendly handling of arrays. Arrays can be created wherever convenient, arrays are efficiently transferred in and out of functions, and arrays are deleted when they are no longer in use.\(^\text{16}\) There is, of course, some efficiency penalty to be paid for having such flexible and easy-to-use arrays. C++ arrays fall somewhere in between; they are as primitive as in Fortran 77, but wrapping them in a class opens up the possibility to quickly create much more user-friendly array objects, and with some significant effort one can implement arrays that mimic those found in Java, Matlab, and Python. C++ also allows you to have good (low-level) control of efficiency, which you do not have in Java, Maple, Matlab,\(^\text{17}\) or Python.\(^\text{18}\)

The convenient array handling and simple plotting capabilities found in Maple, Matlab, and Python make these languages very attractive for explorative scientific computing. One should bear in mind, however, that programs written in Fortran or C/C++ can run much faster.

For exploration of the physical problems and numerical methods covered in most of this book, we think both Matlab and Python represent very productive programming environments. In our opinion, Python has some advantages over Matlab:

- The Python programming language is more powerful
- Several global functions can be placed in a single file (a complete toolbox/module can be contained in one file)
- Transferring functions as arguments to functions is simpler
- Nested, heterogeneous data structures are simple to construct and use
- Object-oriented programming is more convenient
- Interfacing C, C++, and Fortran code is better supported
- Scalar functions work with vectors to a larger extent (without modifications)
- The source is free and runs on more platforms

Having said this, we must add that Matlab has a significantly more comprehensive numerical functionality than Python (linear algebra, ODE solvers, optimization,

\(^{16}\)All this functionality can in fact be implemented in Fortran 77, but the language does not support the functionality, so one has to write lots of low-level code that simulates memory management in a long fixed-size array, reflecting the computer’s memory.

\(^{17}\)In Matlab, most of the array processing functionality takes place in highly optimized Fortran 77 and C functions. Just avoid explicit Matlab loops, see Sect. 6.3.7.

\(^{18}\)With Python it is very easy to migrate array handling code to Fortran 77, C, or C++ to achieve this low-level control of efficiency.
time series analysis, image analysis, etc.) The graphical capabilities of Matlab are also more convenient than those of Python, since Python graphics rely on external packages that must be installed separately.

There is an interface pymat that allows Python programs to use Matlab as a computational and graphics engine. Thus, we prefer to use Python as the programming platform, calling up Matlab when needed.

6.5 Numerical Software Engineering

Numerical software engineering deals with methods for structuring a large piece of numerical software such that it is easy to reuse and maintain the code. Software engineering in general is a huge area of computer science. Many of the methods developed in software engineering are, of course, useful for numerical software as well, but numerical software has certain features that make special demands on software engineering techniques. Contrary to most common non-numerical codes, one often encounters huge data structures, very long execution times, parallel computations, and large amounts of computed numbers whose accuracy can be difficult to verify. The codes can also be very large and complicated. It therefore makes sense to use the term numerical software engineering for software engineering adapted to the world of scientific computing.

6.5.1 Function Libraries

The classic way of organizing numerical codes is to implement algorithms as subroutines in Fortran and collect these subroutines in libraries. An application solving a particular scientific computing problem would then define a set of arrays and other variables and call the appropriate subroutines to get the job done.

The term subroutine is associated with Fortran. When using most other programming languages one would probably refer to a function library.

Numerical Integration Library

Suppose you need to write a piece of code to integrate a function, say $f(x) = x \tanh x$ from 0 to 10, by a suitable numerical integration rule. Choosing the trapezoidal rule as the method, the minimum amount of code would look like

\[
\begin{align*}
n &= 200 \\
s &= 0 \\
x &= 0 \\
&\text{for } i = 1, \ldots, n - 1 \\
x &\leftarrow x + h
\end{align*}
\]
\begin{align*}
s & \leftarrow s + x \tanh x \\
s & \leftarrow h(s + 0.5 \cdot 0 \tanh 0 + 0.5 \cdot 10 \tanh 10)
\end{align*}

Although the code is simple and to the point, it is not reusable on other occasions, i.e., when you want to integrate a new function by the trapezoidal rule. You will then need to write and test the integration method once more. This is a waste of effort – perhaps not that much in the present example, but this is the simplest of all examples. We believe that adopting sound programming styles is a good habit for obtaining a reliable and efficient way of solving problems, even in simple problems where any approach can work well.

The trapezoidal rule should, naturally, be implemented as indicated in all our algorithms from Sect. 6.1.2, namely as a \texttt{Trapezoidal} function (or subroutine) where as many parameters as possible are represented by variables. This means that you should never hardcode the function \( f(x) \) and the integration limits \( a \) and \( b \) as we did above. As soon as the function is well tested, it can be reused as a working “black box” later. If you call this piece of software to integrate a function and see that the result is wrong, you can be quite sure that the error is outside the \texttt{Trapezoidal} function; it must be in the calling code. This principle of breaking up a code into reusable and well-tested pieces is crucial for the efficient and reliable development of software for more complicated problems.

As you need more sophisticated methods for numerical integration, you can implement these as reusable functions. After some time you will have a collection of functions for numerical integration. At that point you need to organize these functions such that you can call the methods you want in an application in a convenient way. The interface, i.e. the argument list, to the functions should be the same, if possible. Furthermore, the functions should be available in a library such that there is no need to copy the complete function codes to your application. The latter point has two important advantages: It keeps the application code small, and there is only one version (code) of the numerical integration function.

Suppose your collection of functions for numerical integration counts four methods:

- \texttt{Trapezoidal}(\(a, b, f, n\))
- \texttt{Simpson}(\(a, b, f, n\))
- \texttt{GaussLegendre}(\(a, b, f, n\))
- \texttt{GaussLobatto}(\(a, b, f, n\))

These functions have the same list of arguments. More sophisticated integration methods, e.g. adaptive schemes, may need more input than \(a, b, f,\) and \(n\) (see page 260).

To make a library, we collect the code associated with the functions in a single file or a collection of files (the file structure can depend on the programming language). None of these files should contain a main program, only functions or subroutines.

Integrating a function numerically, utilizing this library, consists of writing a program doing three things:

1. Enabling access to functionality in the library
2. Implementing the function \(f(x)\) to be integrated
3. Setting \( a, b, \) and \( n \), calling the numerical integration function, and printing the result

We will present small applications running a loop over different \( n \) values and calling up the four methods for each value of \( n \).

Libraries and Compiled Languages

How you technically create the library depends on the programming language used to implement the algorithms. In Fortran, C, and C++ you must compile the library file and place it in a directory that acts as a repository for your libraries. Most real-life libraries are made up of many files, since it would naturally be inconvenient to put all the library code in one big file. These individual files must be compiled, one by one, and then a tool for merging the compiled files into one library file must be invoked. Here is a typical manual procedure, in a Unix environment, for compiling three Fortran 77 files \texttt{file1.f}, \texttt{file2.f}, and \texttt{file3.f} and making a library \texttt{mylib} out of them:

```
unix> f77 -O3 -c file1.f file2.f file3.f
unix> ar libmylib.a file1.o file2.o file3.o
```

The first line compiles the files, resulting in three object files \texttt{file1.o}, \texttt{file2.o}, and \texttt{file3.o}. The second line runs the \texttt{ar} utility to merge the object files into a library file \texttt{libmylib.a}. The name of the library file must always start with \texttt{lib}, and this is not a part of the library name (just \texttt{mylib} is the library name in this case). There are many variants of this theme; the purpose here is to give a brief description of how classic subroutine libraries are created. The library file \texttt{libmylib.a} must be located in a directory for libraries. On a Unix system, \texttt{/usr/lib} is such a directory, but only system administrators have write access to this directory, so one probably needs to create one’s own library directory, e.g., \texttt{$HOME/lib}. In the following we assume that \texttt{libmylib.a} is located in \texttt{$HOME/lib}.

An application can now be written that calls up functionality in the library:

```fortran
real*8 function g(x)
  real*8 x
  g = x*tanh(x)
  return
end

real*8 a, b, I
integer n

program test
  a = 0
  b = 10
  n = 200
  I = trapezoidal(a, b, g, n)
  write(*,*) I
end
```
This code is placed in a file, say, gint.f. The file must be compiled, resulting in the object file gint.o. This object file must thereafter be linked with the library file. The typical manual steps on a Unix computer read

```
unix> f77 -O3 -c gint.f
unix> f77 -L$HOME/lib -o app gint.o -lmylib
```

The first command is the compilation step. The second command is the linking step. The -L option tells f77 where to search for library files, and -o specifies the name of the executable (here app, for application). The rest of the arguments are object files and library files that should be linked together to form the executable. Observe that library files are written with a special syntax, a prefix -l (for library) followed by the name of the library.

C and C++ libraries are created in the same way as explained for Fortran libraries.

Java Libraries

Java libraries consist of classes, and each publicly available class is placed in a file. These files are conveniently organized in directory hierarchies. In the present example with numerical integration, we would typically implement the integration methods as classes Trapezoidal, Simpson, etc., stored, respectively, in files Trapezoidal.java, Simpson.java, and so on. The files are naturally located in a directory, say /home/me/java/int. The application code must import one or more of these methods, and Java searches for classes to import in the directories contained in the CLASSPATH environment variable. In a Bash Unix environment, the CLASSPATH variable is typically initialized in a Bash start-up file .bashrc in your home directory. The initialization statement reads

```
export CLASSPATH=$CLASSPATH:/home/me/java/int
```

There is a utility called jar for packing a set of Java source code files in a single file (jar can be viewed as Java’s counterpart to the Unix tar utility).

Matlab Libraries

Matlab functions are stored in so-called M-files with extension .m. One M-file defines a single function to be used outside the file. For example, the file x.m defines a function x, with as many input arguments and return arguments as desired. Other functions can be placed after the function x in the file x.m, but these functions are only local to the file, so they cannot be accessed from the calling Matlab environment (or script).

In the present example, we would put the various integration methods in separate M-files, i.e., Trapezoidal.m, Simpson.m, and so on. These M-files are conveniently stored in a directory whose name reflects the numerical integration, say /home/me/matlab/int. The name of this directory must be contained in Matlab’s
PATH variable, either by executing

```plaintext
path(path, '/home/me/matlab/int')
```

inside Matlab or by setting the MATLABPATH environment variable in the start-up file. In case your start-up file is .bashrc (Unix Bash environment), the relevant statement becomes

```plaintext
export MATLABPATH=$MATLABPATH:/home/me/matlab/int
```

The current working directory is always in Matlab’s search path.

To summarize, one collects each Matlab library function to be offered to users in a separate M-file, one organizes these M-files in a suitable directory structure, and Matlab’s search path must be updated so that Matlab searches these directories for M-files.

Python Libraries

Python libraries are composed by modules and packages. A module is a file with Python functions and classes, whereas a package is a collection of modules, often with a tree structure (actually reflecting the organization of modules files in a directory tree).

To make a Python module, just place the desired functions in a file. In our example on numerical integration functions, we collect the functions in one file, say, numint.py, and place the file in a directory where Python looks for modules.

The main program for experimenting with different values of \( n \) and different methods can look like this:

```python
#!/usr/bin/env python
import numint  # give access to library for numerical integration
from math import *

def f1(x):
    return exp(-x*x)*log(1+x*sin(x))

a = 0; b = 2
n_values = (10, 100, 1000, 10000)
for n in n_values:
    for method in (Trapezoidal, Simpson, GaussLegendre, GaussLobatto):
        result = method(a, b, f1, n)
        print method.__name__, "\: ", result
```

In this code example, method is a function. We can run through a list of functions, call an entry, and write the name of an entry. This flexibility is not found in the other languages we address in this chapter. As a result, Python makes code organization and experimentation very simple and convenient.

The import statement causes Python to search for a file numint.py in a set of directories specified by the internal Python variable sys.path or the environment variable PYTHONPATH. Suppose you stored numint.py in the directory
6.5 Numerical Software Engineering

/home/me/python/modules. To notify Python about this directory, you can either add the directory name to sys.path in the application code<sup>19</sup>

```python
sys.path = sys.path + ['/home/me/python/modules']
```
or you can set PYTHONPATH appropriately in your start-up file. If you work in a Unix Bash environment, the start-up file is normally .bashrc, and in this file you can write

```bash
export PYTHONPATH=$PYTHONPATH:/home/me/python/modules
```

Maple Libraries

A collection of Maple procedures can be turned into a package, which is the Maple terminology for what we previously have referred to as library. The procedure for creating a package is a bit more comprehensive than for the other programming languages we touch upon here, so we refer to the online help facility in Maple for complete information. Try first help – topic search – package, structure. A complete example on creating a package is found under the main chapter Example Worksheets in the help menu. Go further to Language and System and then to examples/binarytree.

Say the name of your package is numint. Here is a typical use of the package:

```maple
with(numint); # import functions like Trapezoidal, Simpson, ...
f1 := x -> exp(-x*x)*log(1+x*sin(x)); # func to be integrated
qt := Trapezoidal(0, 2, f1, 1000):
qs := Simpson(0, 2, f1, 1000):
printf("Trapezoidal=%e, Simpson=%e", qt, qs);
```

6.5.2 Motivation for Object-Oriented Libraries

The function libraries presented in the previous section are easy to apply; a user just needs to get access to the library and call the desired function. Application codes often require the user to specify at run time the numerical methods to be used for, e.g., integration. This is easily accomplished by reading information about the method and calling the appropriate function in an if–else test:

```maple
if method eq. "Trapezoidal" then
    Trapezoidal(a, b, f, n)
else if method eq. "Simpson" then
    Simpson(a, b, f, n)
```

and so on.

With Python we can build code at run time and execute a string of code with the `exec` function. This allows reading both the method and the function as user input,

<sup>19</sup> Here `sys.path` is a list of strings (directory names), and the statement adds two lists.
and performing the integration with the desired method and function with just one line of code (!):

\[
\text{exec("r = \%s(a, b, lambda x: \%s, n)" % (method, function_expression))}
\]

The integration result is stored in the variable \( r \). The \texttt{lambda} construction is a way of defining a type of inline function. More straight and less compact code could be a multi-line string such as

\[
\text{exec("}
\begin{align*}
\text{def myf(x):} \\
\text{\quad return \%s} \\
\text{r = \%s(a, b, myf, n)} \\
\end{align*}
\text{")}
\]

Of course, this construction of code at run time works only if all integration functions have the same set of parameters, here \( a, b, f, \) and \( n \). This may not be the case when we extend the libraries with more sophisticated integration methods. For example, we can think of methods producing a result with an error less than a specified tolerance. Such methods can take the tolerance and additional parameters, specifying numerical details (such as the order of the integration scheme) as input arguments. A flexible library for numerical integration could have a method collection such as the following:

- Trapezoidal \((a, b, f, n)\)
- Simpson \((a, b, f, n)\)
- AdaptiveTrapezoidal \((a, b, f, n_{\text{max}}, \epsilon)\)
- AdaptiveSimpson \((a, b, f, n_{\text{max}}, \epsilon)\)
- AdaptiveGaussLegendre \((a, b, f, n_{\text{max}}, \epsilon, p_1, p_2)\)
- GeneralGaussLobatto \((a, b, f, n, q_1, q_2, q_3)\)

Here we have just listed some functions for illustrating that the number and type of arguments differ among the functions.

In the if–else approach it is easy to cope with differing argument lists in the function calls, since each call is written explicitly the way it is required. However, in large codes if–else tests tend to be long, and the same tests are often repeated in many places in the code. This assertion may be less easy to realize from the present simple integration example, but the authors’ experience with building large numerical codes points to repeated if–else lists as a major problem. When a new integration method is added to the library, the programmer must remember to add a call to the new function in all relevant if–else tests throughout the code. If the numerical software counts some hundred thousand lines of code, this is a tedious and error-prone task. We shall therefore present a better way of implementing flexible choices of numerical algorithms.

Every time we need to integrate a function in the code, we would like to issue a statement where we do not need to see what type of method that is being used or what type of parameter is needed. The advantage of such an approach is obvious: All calls to numerical integration routines will work with a new method when we have
added that new method to the library. This is a key point in handling large pieces of numerical software. One solution is to employ *object-oriented programming*. In the following we assume that the reader is familiar with object-oriented programming from a basic course in Java or C++.

In object-oriented programming we represent the numerical algorithms by *objects*. Objects contain both data and functions (operating on the object data and external data). We introduce a pseudo-code notation for objects:

```java
class Trapezoidal
data:
    a, b, f, n
methods:
    integrate()
    init(a, b, f, n)
```

This pseudo code defines the contents of objects of type `Trapezoidal`. Each such object will contain the data `a`, `b`, `f`, and `n` (with obvious meaning in the present example) and a function called `integrate`, for carrying out the numerical integration, and another function `init`, for initializing the data members.

We create a class for each integration method, say, `Trapezoidal`, `Simpson`, and so on. Each class is *derived* from a general base (or super) class called `Integration`. The base class defines the set of legal methods that can be applied to all integration methods and it also declares the set of data that can be shared among various integration methods (typically `a`, `b`, `f`, and `n` in our example). As soon as we have created an object representing a particular method, say, `Simpson`, we can treat this object as being of type `Integration` in the rest of the code. This means that we hide the information that we are working with Simpson’s rule; all we can see is a general numerical integration object. When we want to integrate a function, we have an object (say) `i` and just call `i`'s `integrate` method, without arguments. The magic of object-oriented programming is that the program knows the type of integration object we actually have (say, `Simpson`) and calls the `integrate` method in the appropriate subclass (here `Simpson`'s `integrate` method). Since the object type can be treated as “identical” for all methods, and since all `integrate` methods take the same argument list (empty in this case), the application code for integrating a function is independent of the method we use.

At this stage it might be encouraging to realize that the outlined object-oriented programming approach has wide applications. Numerical software is basically a collection of different types of numerical methods (integration, the solution of ODEs, the solution of nonlinear equations, the solution of linear systems of equations, etc.) and a “main program” calling up a combination of methods to solve a specific problem. A certain type of numerical method has many different algorithms. Employing the idea of object-oriented programming, each type of numerical method is implemented as a class hierarchy, where the different algorithms are realized as different subclasses. In this way, we will have a class hierarchy for numerical integration, ODE solvers, nonlinear equation solvers, and so on. When an ODE solver needs a nonlinear equation solver as part of the algorithm, it can work with a general (base class) interface to all methods for nonlinear equations. The code applying these methods does not need to distinguish between bisection or Newton’s method; just
the nonlinear equation solver base class and its generic (say) solve method are
seen. This way of programming hides details in numerical algorithms and reduces
complexity. This is of course very important for the reliability of large numerical
codes.

6.5.3 Design of a Numerical Integration Hierarchy

Class **Integrate** is the base class of our numerical integration methods hierarchy. Specific integration rules are implemented as subclasses. Common to all methods are the integration limits and the function to be integrated. Other parameters can differ among the subclasses. The idea of an init method with differing argument lists is not good. We should look for a strategy that handles input data and the initialization of integration objects in the same way for all subclasses of **Integration**. To achieve a unified treatment of input data for all subclasses, it may be convenient to collect all the parameters in a so-called parameter class, here called **Integration_prm**:

```python
class Integration_prm:
    data:
        a, b, f
        npoints
        method
tolerance
    ...
methods:
        constructor()
        create()
        default()
        read()
        write()
```

This outline of the class shows some possible parameters and (at least) five methods:

- **constructor** for creating and initializing an object
- **create** for creating an integration rule object (as a subclass instance in the **Integration** hierarchy), based on the data in this **Integration_prm** instance
- **default** for assigning appropriate default values to data members
- **read** for reading values from some kind of user interface
- **write** for dumping the contents of the parameters (for a check)

Observe that our pseudo code does not coincide with a particular programming language.\(^\text{20}\)

The base class **Integration** for all integration methods typically has a constructor that takes the bag of parameters (i.e., an **Integration_prm** object) and stores it internally in the class. The base class also offers access to the **integrate** method

---

\(^{20}\) For example, the constructor method will have different names in different languages (**Integration_prm** in Java and C++, **__init__** in Python).
common to all subclasses, where the numerical integration rule is to be implemented (in the subclasses):

```java
class Integration
    data:
        prm
    methods:
        constructor(Integration_prm p)
            prm = p
        integrate()
```

Whenever appropriate, we insert the body of a function as indented statements under the function heading.

The trapezoidal method is our first subclass candidate:

```java
class Trapezoidal, subclass of Integration
data:
    methods:
        constructor(Integration_prm p)
            Integration.constructor(p)
        integrate()
            h = (prm.b-prm.a)/(prm.npoints-1)
            x = a
            s = 0
            for i = 2,...,n-1
                x = x + h
                s = s + f(x)
            s = s + 0.5*(f(prm.a) + f(prm.b))
            return h*s
```

This class apparently has data on its own, but it inherits the `prm` object from its base class `Integration`, and no additional data are needed. The `constructor` passes the `Integration_prm` input object onto the base class constructor, which stores the parameter object as the data member `prm`. In the `integrate` method, the integration parameters are accessed through the `prm` object. For example, `prm.a` means the `a` data member of object `prm`. This is the same notation as used in common languages supporting classes, such as C++, Java, and Python.

### 6.5.4 A Class Hierarchy in Java

Readers who are familiar with the Java programming language will quite quickly convert the previous pseudo code to Java. The following text is written for this group of readers.

The base class `Integration` is realized as

```java
class Integration {
    public Integration_prm prm;

    public Integration (Integration_prm p) {
        prm = p;
    }

    public double integrate () {
    ```
A specific integration method, such as the trapezoidal rule, is implemented in the subclass `Trapezoidal`:

```java
class Trapezoidal extends Integration {
    public Trapezoidal (Integration_prm p) {
        super(p);
    }

    public double integrate () {
        double a = prm.a; double b = prm.b; int n = prm.n;
        Func f = prm.f;

        double h = (b-a)/((double)n);
        double s = 0;
        double x = a;
        int i;
        for (i = 1; i <= n-1; i++) {
            x = x + h;
            s = s + f.f(x);
        }
        s = 0.5*(f.f(a) + f.f(b)) + s;
        return h*s;
    }
}
```

The `Integration_prm` class holds `a`, `b`, `n`, and `f`. In addition, the class provides a function `read` for extracting information about `a`, `b`, and `n` from command-line arguments. A function `write` dumps the contents of `a`, `b`, and `n` on the screen. Here is the code:

```java
class Integration_prm {
    public double a, b;
    public Func f;
    public int n;
    public String method;

    public Integration_prm () {
        /* default values */
        a = 0; b = 1; n = 10; f = new f2();
        method = "Trapezoidal";
    }

    public void read (String argv[]) {
        int i;
        for (i = 0; i < argv.length; i=i+2) {
            if (argv[i].compareTo("-a") == 0) {
                a = Double.valueOf(argv[i+1]).doubleValue();
            }
            if (argv[i].compareTo("-b") == 0) {
                b = Double.valueOf(argv[i+1]).doubleValue();
            }
            if (argv[i].compareTo("-n") == 0) {
                n = Integer.valueOf(argv[i+1]).intValue();
            }
            if (argv[i].compareTo("-m") == 0) {
```
method = argv[i+1];
}
}

public void write ()
{
    System.out.println("a=" + a + " b=" + b + " n=" + n + " method=" + method);
}

public Integration create ()
{
    Integration i;
    if (method.compareTo("Trapezoidal") == 0) {
        i = new Trapezoidal(this);
    } // else if (method.compareTo("Simpson") == 0) and so on...
    else {
        i = new Trapezoidal(this);
    }
    return i;
}

The specification of functions to be integrated is exactly the same as in Sect. 6.3.4; that is, \( f(x) \) must be programmed as a subclass of Func, since Java does not allow stand-alone functions.

The final piece of the Java code is the main program, here a part of a class called Demo for testing the implementation:

class Demo {
    public static void main (String argv[]) {
        Integration_prm p = new Integration_prm();
        p.read(argv);
        p.write();
        f1 f = new f1();
        Integration i = p.create();
        double result=0; int j;
        for (j = 1; j <= 10000; j++) {
            result = i.integrate();
        }
        System.out.println(result);
    }
}

Notice that the variable \( i \) is of type Integration and that \( i\text{.integrate} \) is correctly interpreted as a call to the integrate method in the Trapezoidal class.

### 6.5.5 A Class Hierarchy in Python

Let us show how the Integration hierarchy can be implemented in the Python programming language. We exemplify one subclass, Trapezoidal, and with this information, the reader should be able to construct other subclasses (Simpson, for instance). The text here assumes that you are familiar with several Python topics:
class programming, overloaded operators in classes, type checking, and `eval/exec`. A suitable background can be gained by looking up these keywords in the index of H. P. Langtangen’s *Scripting Tools for Scientific Computations*.

We suppose that we have created a class `Integrationprm` (details follow later). The base class for all integration methods can then be implemented as the Python class `Integration`.

```python
class Integration:
    """
    Base class for numerical methods for integrating
    f(x) over [a,b].
    """

def __init__(self, p):
    self.prm = p  # Integration_prm object

def integrate(self):
    """Perform integration.""
    # to be implemented in subclasses
    return "Integration.integrate; not impl. in subclass"
```

The trapezoidal rule is conveniently realized as a subclass:

```python
class Trapezoidal(Integration):
    def __init__(self, p):
        Integration.__init__(self, p)

    def integrate(self):
        p = self.prm; a = p.a; b = p.b; n = p.npoints; f = p.f
        h = (b-a)/float(n)
        s = 0
        x = a
        for i in range(1,n,1):
            x = x + h
            s = s + f(x)
        s = 0.5*(f(a) + f(b)) + s
        return h*s
```

Other subclasses implement other integration rules.

The `Integrationprm` class just holds `a`, `b`, `n`, and `f`, but it is convenient to supply the class with functionality for setting default values and for reading information from the command line. To set the function `f`, we use a utility `Function`, to be explained later. For now it is sufficient to know that

```python
Function('sin(x)+1', independent_variable='x')
```

defines a function of `x` equal to `sin(x)+1`. In other words, the `Function` utility allows us to specify Python functions directly from text expressions (!).

Convenient reading of command-line parameters is done with a function `cmldict`, which returns all command-line options as a dictionary (say) `p`, such that `p[option]=value`. That is, `-a 0.1` on the command-line results in `p['a']` being 0.1. Alternatively, we can traverse the array of command-line strings as we did in the Java code. Here is the code for the parameter class:

```python
class Integrationprm:
    """
    Holds all parameters needed to initialize objects
    """
```
in the Integration hierarchy.

```python
def __init__(self):
    self.default()
    return

def default(self):
    """Assign appropriate default values to all parameters."""
    self.a = 0
    self.b = 1
    self.npoints = 10
    # default f(x)=1+x:
    self.f = Function('1+x', independent_variable='x')
    self.method = "Trapezoidal"
    # alternative: could use a dictionary and __getitem__

def read(self, argv=sys.argv[1:], short_opt="", long_opt=""):  
    """Read from the command line."""
    from cmldict import cmldict
    defaults = { 'a' : 0, 'b' : 1, 'n' : 10,
                'f' : Function("1+x"), 'm' : "Trapezoidal" }
    p = cmldict(argv, cmlargs=defaults, validity=0)
    self.a = p['a']
    self.b = p['b']
    self.f = Function(p['f'])
    self.npoints = p['n']
    self.method = p['m']

    # alternative manual parsing by traversing argv:
    for i in range(0,len(argv),2):
        if argv[i] == "-a": self.a = float(argv[i+1])
        if argv[i] == "-b": self.b = float(argv[i+1])
        if argv[i] == "-n": self.n = int(argv[i+1])
        if argv[i] == "-f": self.f = Function(argv[i+1])
        if argv[i] == "-m": self.method = argv[i+1]

    def write(self):
        print "a=%g b=%g f='%s' n=%d method='%s' % % 
            (self.a, self.b, self.f.__name__, \
            self.npoints, self.method)

    def create(self):
        """Create subclass of Integration""
        code = "i = %s(self)" % self.method
        exec(code)  # turn string into Python code
        return i
```

Note that the `create` function is very simple, because the string in `self.method` is supposed to coincide with a subclass name in the `Integration` hierarchy. We can therefore construct a string `code` containing the construction of the proper subclass instance and use `exec` to execute this code segment. Alternatively, we could have used an if–else test, as we did in the corresponding `create` method in the Java code.

Here is an example on a typical main program using an `Integration_prm` instance and an `Integration` subclass instance to evaluate an integral:

```python
p = Integration_prm()
p.read(sys.argv[1:])
p.write()
i = p.create()
```
result = i.integrate()
print result

Running such a type of Python program with

```
-a 0 -b 2 -m Trapezoidal -f '1+x*x*x'
```

as command-line arguments gives the result 6.04.

The details of the `Function` class are not important, but the code is listed here for reference and to demonstrate how easy it is with Python to build a convenient utility for representing functions. This utility can be used to hold either Python functions or string definitions of functions.\(^{21}\)

```python
class Function:
    ""
    Unified treatment of functions; strings or function objects.
    Examples on usage:
    ```
    def myfunc(x):
        return 1+x
    f = Function(myfunc, 'x')  # attach Python function
    v = f(1.2)
    f = Function('1+t', 't')  # specify function by string
    v = f(1.2)
    ""
    def __init__(self, f, independent_variable='x'):
        self.f = f  # expression or function object
        self.var = independent_variable  # 'x', 't' etc.
        if type(f) == type(''):  # string:
            self.f_is_string = 1
            self.__name__ = self.f
        else:
            # function object:
            self.f_is_string = 0
            self.__name__ = self.f.__name__
    def __call__(self, x):
        if self.f_is_string:
            exec("%s = %g" % (self.var, x))
        return eval(self.f)
        else:
            return self.f(x)
```

6.5.6 Object-Oriented Programming in Matlab

The Matlab language supports object-oriented programming. Nevertheless, these authors find it is more convenient to perform object-oriented programming in languages such as Java, C++, and Python. These languages were originally designed

\(^{21}\)Class `Function` implements the flexibility of the `fcnchk` function in Matlab, but results in nicer call syntax (functions represented as `Function` instances are called like any other Python functions).
for object-oriented programming, while Matlab offers this programming technique as a more recent add-on to the language. On the other hand, Matlab allows the use of Java classes mixed with Matlab statements. This is a convenient way of introducing object-oriented programming in Matlab code. In our example, we could reuse the Java classes Integrationprm, Integration, Trapezoidal and so on, from Sect. 6.5.4, but we shall not go further into this topic here.

6.6 Exercises

Exercise 6.1. Write a mathematical pseudo code for solving the quadratic algebraic equation

$$ax^2 + bx + c = 0.$$ 

The algorithm should take $a$, $b$, and $c$ as input, and return one or two roots.

Exercise 6.2. A straightforward mathematical pseudo code from Exercise 6.1 can fail if $a = 0$. Also, if $a = 0$ and $b = 0$, $c$ must equal zero. Extend the pseudo code with appropriate tests on $a = 0$ and $a = b = 0$.

Exercise 6.3. The roots in the algorithm in Exercise 6.1 can be real or complex, depending on the sign of $b^2 - 4ac$. Many programming languages handle the square roots of negative numbers correctly, i.e., compute the roots as complex numbers. However, some languages do not support complex numbers well, and the algorithm should in such cases ensure that only real arithmetic is involved. Modify the pseudo code in Exercise 6.1 such that the roots are represented by four real numbers, the real and imaginary parts of the two roots. Make sure that all operations involve real numbers only.

Exercise 6.4. Implement Algorithm 6.6 in a computer program. Find a suitable test problem for verifying the implementation.

Exercise 6.5. Work through Algorithm 6.9 by hand for the choices $a = 0$, $b = 1$, $f(x) = x^3$, and an initial set of points between $x_0 = 0$ and $x_1 = 1$. Let $E$ correspond to the error in the trapezoidal rule (6.12), and set $\varepsilon = 0.01$.

Exercise 6.6. Implement Algorithm 6.9 in a computer program. Find a test problem that can be used to verify the implementation.

Exercise 6.7. Calling the heun function in Algorithm 6.11 $N$ times, integrating from 0 to some time level in each call, was pointed out as a very inefficient way of producing a set of data points $(i \Delta t, u(i \Delta t))$. Find out how you can produce these data points more efficiently, still using the heun function in Algorithm 6.11 (hint: perform one step at a time). Count the number of $f$-function evaluations in the two approaches.
**Exercise 6.8.** Suppose we have a differential equation on the form

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

that is, \( f \) can be an explicit function of \( t \). A simple example is \( f(u, t) = -2t \times u \), with the corresponding solution \( u = e^{-t^2} \).

Understand the idea behind the derivation of Heun’s method, and use this understanding to generalize Algorithm 6.12 such that it works for the right-hand side functions \( f(u, t) \) in the above differential equation.

\[ \hat{=} \]

### 6.7 Projects

#### 6.7.1 Computing the Volume of a Cylindrical Container

The purpose of this project is to compute the volume of water in a cylindrical container. An example of such a container is given in Fig. 6.3. The radius of the container can change along the center axis, and is given by a function \( R(x) \). Water is stored in the container up to a height \( H \). The goal is to develop a program that can compute the water volume in such a container, given the function \( R(x) \) and the parameter \( H \) as input.

![Fig. 6.3 Sketch of a cylindrical container with circular cross section and radius \( R(x) \). The container contains water up to a height \( H \)](image-url)
6.7 Projects

(a) Show that the water volume is given by the expression

\[ V = \int_0^H \int_0^{2\pi} \int_0^{R(x)} r dr d\theta dx. \]  

Here, \((r, \theta, x)\) denotes cylindrical coordinates. Show that the integral (6.14) reduces to

\[ V = \pi \int_0^H [R(x)]^2 dx. \]  

(b) Write the function \(R(x)\) in Fig. 6.3 as mathematical pseudo code.

(c) Write a mathematical pseudo code for calculating \(V\), volume \((R, H, n)\)

given \(R(x), H\), the number of evaluation points \(n\), using the trapezoidal rule.

(d) Implement the algorithm from (c) with the aid of a programming language.

(e) Show how an implementation of Algorithm 6.2 in a specific programming language can be reused as is for computing \(V\). (Hint: Define a function \(f(x)\) as \(\pi [R(x)]^2\), with \(R(x)\) as a global function.) Discuss the pros and cons of this approach and the tailored solution in (d).

6.7.2 A Class Hierarchy for Scalar ODE Solvers

This project concerns applying the ideas of Sect. 6.5.2 to the solution of scalar ODEs. We introduce a base class ODESolver for all solver algorithms, and a class ODESolver_prm holding all parameters needed for initializing and running an ODESolver subclass instance.

(a) Write a pseudo code for class ODESolver_prm, specifying data members and methods.

(b) Write a pseudo code for class ODESolver.

(c) Write a pseudo code for a specific ODE algorithm, such as Heun’s method, as a subclass of ODESolver.

(d) Implement the class hierarchies in Java, C++, and Python.

(e) Find at least three different solution algorithms for the ODEs and implement these algorithms as subclasses of ODESolver.

(f) Make libraries out of the implementation in (e). The C++ library can be named libODE.a, the Java library ODE.jar, and the Python module ODE.py. (See Sect. 6.5.1 for information on how to create such libraries.)

(g) Construct a test problem for verifying the implementation. Write associated main programs and check that the code works. The main programs should make use of the libraries from (f).
6.7.3 Software for Systems of ODEs

This project extends Sect. 6.4 and Project 6.7.2 to systems of ODEs.

(a) Generalize Algorithm 6.12 to systems of ODEs. Make two versions of the algorithm: one that does not store values in an array, except at the final step, and one that stores all the computed values in an array.

(b) Implement the algorithm for solving a system of ODEs as a function in a programming language. Consider only the algorithm that stores the computed values at the final step.
   The other algorithm would need to return a two-dimensional array, with one index for the step numbers and one index for the components in the ODE system. Since two-dimensional arrays are not covered in Sect. 6.4, we drop the corresponding implementation. In the version of the algorithm to be implemented, return the solution as an array, where each element contains the corresponding component in the solution (at the final, $N$-th, step of the algorithm). Note that calling the algorithm repeatedly with $N = 1$ (single step at a time) makes it easy to load the solution at many time steps into some array in the calling code. This array might be convenient for visualizing the solution.

(c) Extend the ODESolver.prn class and the ODESolver hierarchy to systems of ODEs. Use the code from (b) in a subclass of ODESolver.

(d) Construct a test problem and verify the implementations.
Elements of Scientific Computing
Tveito, A.; Langtangen, H.P.; Nielsen, B.F.; Cai, X.
2010, XII, 468 p. 86 illus., Hardcover
ISBN: 978-3-642-11298-0