CHAPTER 1

# Introduction

*Science sans conscience n'est que ruine de l'âme.*[1]

—François Rabelais

Teaching numerical methods was a major discipline of computer science at a time computers were used only by very few professionals such as physicists or operation research technicians. At that time, most of the problems solved with the help of a computer were of numerical nature, such as matrix inversion or optimization of a function with many parameters.

With the advent of minicomputers, workstations, and, foremost, personal computers, the scope of problems solved with a computer shifted from the realm of numerical analysis to that of symbol manipulation. Recently, the main use of a computer has been centered on office automation. Major applications are word processors and database applications.

Today, computers are no longer working stand-alone. Instead, they are sharing information with other computers. Large databases are becoming commonplace. The wealth of information stored in large databases tends to be ignored, mainly because only a few persons know how to get access to it and even fewer know how to extract useful information. People have recently started to tackle this problem under the buzzword *data mining*. In truth, data mining is nothing other than good old numerical data analysis performed by high-energy physicists with the help of computers. Of course, a few new techniques have been invented recently, but most of the field now consists of rediscovering algorithms used in the past. This past goes back to the day Enrico Fermi used the ENIAC to perform phase shift analysis to determine the nature of nuclear forces.

The interesting point, however, is that, with the advent of data mining, numerical methods are back on the scene of information technologies.

1. Science without consciousness just ruins the soul.

## 1.1 Object-Oriented Paradigm and Mathematical Objects

In recent years, object-oriented programming (OOP) has been welcomed for its ability to represent objects from the real world (employees, bank accounts, etc.) inside a computer. Herein resides the formidable leverage of object-oriented programming. It turns out that this way of looking at OOP is somewhat overstated (as these lines are written). Objects manipulated inside an object-oriented program certainly do not behave like their real-world counterparts. Computer objects are only models of those of the real world. The Unified Modeling Language (UML) user guides goes further in stating that "a model is a simplification of reality" and we should emphasize that it is "only that." OOP modeling is so powerful, however, that people tend to forget about it and only think in terms of real-world objects.

An area in which the behavior of computer objects nearly reproduces that of their real-world counterparts is mathematics. Mathematical objects are organized within *hierarchies*. For example, natural integers are included in integers (signed integers), which are included in rational numbers, themselves included in real numbers. Mathematical objects use *polymorphism,* in that one operation can be defined on several entities. For example, addition and multiplication are defined for numbers, vectors, matrices, polynomials—as we shall see in this book—and many other mathematical entities. Common properties can be established as an abstract concept (e.g., a group) without the need to specify a concrete implementation. Such concepts can then be used to prove a given property for a concrete case. All this looks very similar to *class hierarchies*, *methods,* and *inheritance.*

Because of these similarities, OOP offers the possibility to manipulate mathematical objects in such a way that the boundary between real objects and their computer models becomes almost nonexistent. This is no surprise since the structure of OOP objects is equivalent to that of mathematical objects[2]. In numerical evaluations, the equivalence between mathematical objects and computer objects is almost perfect. One notable difference remains, however—namely, the finite size of the representation for noninteger number in a computer limiting the attainable precision. We shall address this important topic in Section 1.3.2.

Most numerical algorithms have been invented long before the widespread use of computers. Algorithms were designed to speed up human computation and therefore were constructed to minimize the number of operations to be carried out by the human operator. Minimizing the number of operations is the best thing to do to speed up code execution.

One of the most heralded benefits of object-oriented programming is code reuse, a consequence, in principle, of the hierarchical structure and inheritance. The last statement is pondered by "in principle" since, to date, code reuse of real-world objects is still far from being commonplace.

2. From the point of view of computer science, OOP objects are considered as mathematical objects.

For all these reasons, this book tries to convince you that using object-oriented programming for numerical evaluations can exploit the mathematical definitions to maximize code reuse between many different algorithms. Such a high degree of reuse yields very concise code. Not surprisingly, this code is quite efficient and, most important, highly maintainable. Providing more than an argumentation, I show how to implement some numerical algorithms selected among those that, in my opinion, are most useful for the areas in which object-oriented software is primarily used: finance, medicine, and decision support.

## 1.2 Object-Oriented Concepts in a Nutshell

First, let us define what is covered by the adjective *object oriented*. Many software vendors are qualifying a piece of software as object oriented as soon as it contains things called objects, even though the behavior of those objects has little to do with object orientation. For many programmers and most software journalists, any software system offering a user interface design tool on which elements can be pasted on a window and linked to some events—even though most of these events are being restricted to user interactions—can be called object oriented. There are several typical examples of such software, all of them having the prefix *visual* in their names[3]. Visual programming is something entirely different from object-oriented programming.

Object-orientation is something different, not intrinsically linked with the user interface. Recently, object-oriented techniques applied to user interfaces have been widely exposed to the public, hence the confusion. Three properties are considered essential for object-oriented software:

1. data encapsulation
2. class hierarchy and inheritance
3. polymorphism

*Data encapsulation* is the fact that each object hides its internal structure from the rest of the system. Data encapsulation is in fact a misnomer since an object usually chooses to expose some of its data. I prefer to use the expression *hiding the implementation*, a more precise description of what is usually understood by data encapsulation. Hiding the implementation is a crucial point because an object, once fully tested, is guaranteed to work ever after. It ensures an easy maintainability of applications because the internal implementation of an object can be modified without impacting the application, as long as the public methods are kept identical.

*Class hierarchy and inheritance* is the keystone implementation of any object-oriented system. A *class* is a description of all properties of all objects of the same type. These properties can be structural (static) or behavioral (dynamic). Static

3. This is not to say that all products bearing a name with the prefix *Visual* are not object-oriented.

properties are mostly described with instance variables. Dynamic properties are described by methods. *Inheritance* is the ability to derive the properties of an object from those of another. The class of the object from which another object is deriving its properties is called the *superclass*. A powerful technique offered by class hierarchy and inheritance is the overloading of some of the behavior of the superclass.

*Polymorphism* is the ability to manipulate objects from different classes, not necessarily related by inheritance, through a common set of methods. To take an example from this book, polynomials can have the same behavior as signed integers with respect to arithmetic operations: addition, subtraction, multiplication, and division.

Most so-called object-oriented development tools (as opposed to languages) usually fail the inheritance and polymorphism requirements.

The code implementation of the algorithms presented in this book is given in two languages: Smalltalk and Java. Both languages are excellent object-oriented languages. I would strongly recommend that readers consult the implementation sections of both languages regardless of their personal taste of language for two reasons: I have tried to make use of the best feature of each language, and each implementation has been made independently. The fact that the code of each implementation is different shows that there are indeed many ways to skin a cat, even when done by the same person. Thus, looking seriously at both implementations can be quite instructive for someone who wants to progress with the object-oriented paradigm.

## 1.3 Dealing with Numerical Data

The numerical methods exposed in this book are all applicable to real numbers. As noted earlier, the finite representation of numbers within a computer limits the precision of numerical results, thereby causing a departure from the ideal world of mathematics. This section discusses issues related to this limitation.

### 1.3.1 Floating Point Representation

Currently humankind is using the decimal system[4]. In this system, however, most rational numbers and all irrational and transcendental numbers escape our way of representation. Numbers such as 1/3 or $\pi$ cannot be written in the decimal system other than approximately. One can choose to add more digits to the right of the decimal point to increase the precision of the representation. The true value of the number, however, cannot be represented. Thus, in general, a real number cannot be

4. This fact is, of course, quite fortuitous. Some civilizations opted for a different base. The Sumerians have used the base 60 systems, and this habit has survived until now in our time units. The Maya civilization was using the base 20. The reader interested in the history of numbers ought to read the books of Georges Ifrah [Ifrah].

represented by a finite decimal representation. This kind of limitation has nothing to do with the use of computers. For example, ancient civilizations constantly struggled to get an exact determination of the number $\pi$. To go around that limitation, mathematicians have invented abstract representations of numbers, which can be manipulated in regular computations. This includes irreducible fractions (e.g.,1/3), irrational numbers ($\sqrt{2}$ e.g.), transcendental numbers ($\pi$ and $e$ the base of natural logarithms e.g.), and normal[5] infinities ($-\infty$ and $+\infty$).

Like humans, computers are using a representation with a finite number of digits, but the digits are restricted to 0 and 1. Otherwise, number representation in a computer can be compared to the way we represent numbers in writing. Compared to humans computers have the notable difference that the number of digits used to represent a number cannot be adjusted during a computation. There is no such thing as adding a few more decimal digits to increase precision. One should note that this is only an implementation choice. One could think of designing a computer manipulating numbers with adjustable precision. Of course, some protection should be built in to prevent a number, such as 1/3, to expand ad infinitum. Probably, such a computer would be much slower. In digital representation (the word *digital* being taken in its first sense—that is, a representation with digits), no matter how clever the implementation[6], most numbers will always escape the possibility of exact representation.

In present-day computers, a floating-point number is represented as $m \times r^e$, where the radix $r$ is a fixed number, generally 2. On some machines, however, the radix can be 10 or 16. Thus, each floating-point number is represented in two parts[7]: an integral part called the mantissa $m$ and an exponent $e$. This approach is quite familiar to people using large quantities (e.g., astronomers) or studying the microscopic world (e.g., microbiologists). Of course, the natural radix for people is 10. For example, the average distance from Earth to the sun expressed in kilometers is written as $1.4959787 \times 10^8$.

In the case of radix 2, the number 18446744073709551616 is represented as $1 \times 2^{64}$. Quite a shorthand compared to the decimal notation! IEEE standard floating-point numbers use 24 bits for the mantissa (about 8 decimal digits) in single precision; they use 53 bits (about 15 decimal digits) in double precision.

One important property of floating-point number representation is that the relative precision of the representation—that is, the ratio between the precision and the number itself—is the same for all numbers except, of course, for the number 0.

---

5. Since Cantor, mathematicians have learned that many kinds of infinities exist. See, for example, reference [Gullberg].

6. Symbolic manipulation programs do represent numbers as we do in mathematics. Such programs are not yet suited for quick numerical computation, but research in this area is still open.

7. This statement is admittedly a simplification. In practice, exponents of floating-point numbers are offset to allow negative exponents. This does not change the point being made in this section, however.

### 1.3.2 Rounding Errors

To investigate the problem of rounding, let us use our own decimal system, limiting ourselves to 15 digits and an exponent. In this system, the number $2^{64}$ is now written as $184467440737095 \times 10^5$. Let us now perform some elementary arithmetic operations.

First, many people are aware of problems occurring with addition or subtraction. Indeed, we have

$$184467440737095 \times 10^5 + 300 = 184467440737095 \times 10^5.$$

More generally, adding or subtracting to $2^{64}$ any number smaller than 100000 is simply ignored by our representation. This is called a *rounding error*. This kind of rounding error has the nontrivial consequence of breaking the associative law of addition. For example,

$$\left(1 \times 2^{64} + 1 \times 2^{16}\right) + 1 \times 2^{32} = 184467440780044 \times 10^5,$$

whereas

$$1 \times 2^{64} + \left(1 \times 2^{16} + 1 \times 2^{32}\right) = 184467440780045 \times 10^5.$$

In the two last expressions, the operation within the parentheses is performed first and rounded to the precision of our representation, as this is done within the floating-point arithmetic unit of a microprocessor[8].

Other types of rounding errors may also occur with factors. Translating the calculation $1 \times 2^{64} \div 1 \times 2^{16} = 1 \times 2^{48}$ into our representation yields

$$184467440737095 \times 10^5 \div 65536 = 2814744976710655.$$

The result is off by just one since $2^{48} = 2814744976710656$. This seems not to be a big deal since the relative error—that is, the ratio between the error and the result—is about $3.6 \times 10^{-16}\%$.

Computing $1 \times 2^{48} - 1 \times 2^{64} \div 1 \times 2^{16}$, however, yields $-1$ instead of 0. This time the relative error is 100% or infinite depending on what reference is taken to compute the relative error. Now, imagine that this last expression was used in finding the real (as opposed to complex) solutions of the second order equation:

$$2^{-16}x^2 + 2^{25}x + 2^{64} = 0.$$

The solution to that equation is

$$x = \frac{-2^{24} \pm \sqrt{2^{48} - 2^{64} \times 2^{-16}}}{2^{-16}}.$$

8. In modern microprocessors, a floating-point arithmetic unit actually uses more digits than the representation. These extra digits are called *guard digits*. Such difference is not relevant for our example.

Here, the rounding error prevents the square root from being evaluated since $\sqrt{-1}$ cannot be represented as a floating-point number. Thus, it has the devastating effect of transforming a result into something, that cannot be computed at all.

This simplistic example shows that rounding errors, however harmless they might seem, can have quite severe consequences. An interested reader can reproduce these results using the Smalltalk class described in Appendix ???.

In addition to rounding errors of the kind illustrated so far, rounding errors propagate in the computation. Study of error propagation is a wide area beyond the scope of this book. This section is only meant as a reminder that numerical results coming out of a computer must always be taken with a grain of salt. The only good advice to give at this point is to try the algorithm out and compare the changes caused by small variations of the inputs over their expected range. There is no shame in trying things out, and you will avoid the ridicule of someone proving that your results are nonsense.

The interested reader will find a wealth of information about floating-number representations and their limitations in the book of Donald E. Knuth [Knudth 2]. An excellent article by David Goldberg is recommended for a quick but in-depth survey: "*What Every Computer Scientist Should Know About Floating Point Arithmetic*," published in the March 1991 issue of *Computing Surveys* and available from various Web sites. Let us conclude this section with a quotation from Knuth [Knudth 2]:

> Floating point arithmetic is by nature inexact, and it is not difficult to misuse it so that the computed answers consist almost entirely of "noise." One of the principal problems of numerical analysis is to determine how accurate the results of certain numerical methods will be.

### 1.3.3 Real Example of Rounding Error

To illustrate how rounding errors propagate, let us work our way through an example. Let us consider a numerical problem whose solution is known, that is, the solution can be computed exactly.

This numerical problem has one parameter, that measures the complexity of the data. Moreover, data can be of two types: general data or special data. Special data have some symmetry properties, that can be exploited by the algorithm. Let us now consider two algorithms, A and B, able to solve the problem. In general, algorithm B is faster than algorithm A.

The precision of each algorithm is determined by computing the deviation of the solution given by the algorithm with the value known theoretically. The precision has been determined for each set of data and for several values of the parameter measuring the complexity of the data.

Figure 1.1 shows the results. The parameter measuring the complexity is laid on the $x$-axis using a logarithmic scale. The precision is expressed as the negative of the decimal logarithm of the deviation from the known solution. The higher the value the better is the precision. The precision of the floating-point numbers on the machine used in this study corresponds roughly to 16 on the scale of Figure 1.1.

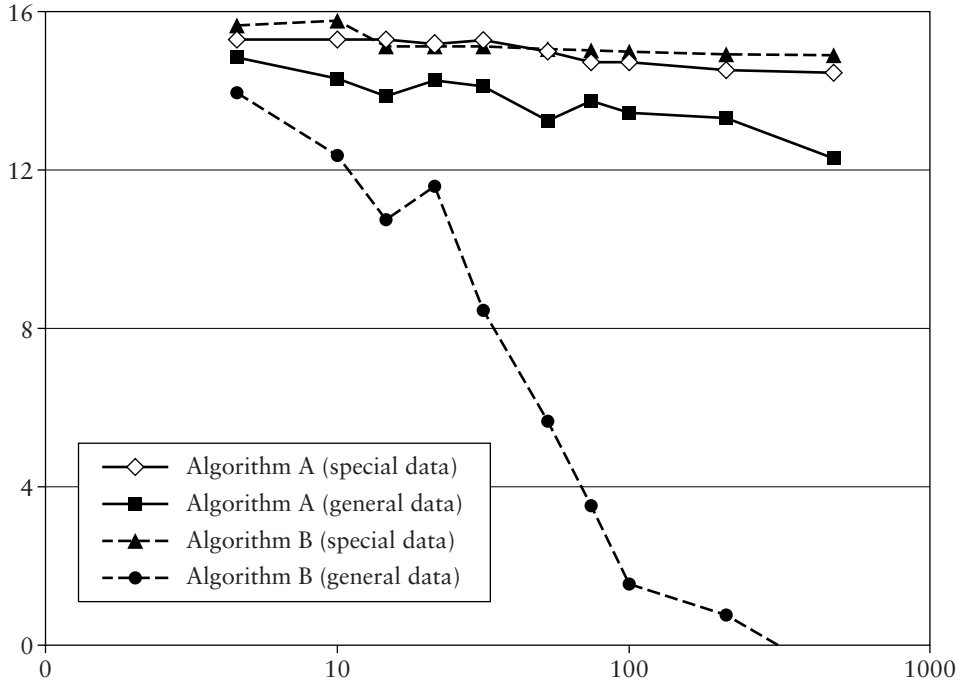


**FIG. 1.1** Comparison of achieved precision

The first observation does not come as a surprise: the precision of each algorithm degrades as the complexity of the problem increases. One can see that when the algorithms can exploit the symmetry properties of the data, the precision is better (curves for special data) than for general data. In this case, the two algorithms are performing with essentially the same precision. Thus, one can choose the faster algorithm—namely, algorithm B. For the general data, however, algorithm B has poorer and poorer precision as the complexity increases. For complexity larger than 50, algorithm B becomes totally unreliable, to the point of becoming a perfect illustration of Knuth's quotation earlier. Thus, for general data, one has no choice but to use algorithm A.

Readers who do not like mysteries can read Section ??? where these algorithms are discussed.

### 1.3.4 Outsmarting Rounding Errors

In some instances, rounding errors can be significantly reduced if one spends some time reconsidering how to compute the final solution. This section presents an example of such thinking.

Consider the following second-order equation, which must be solved when looking for the eigenvalues of a symmetrical matrix (see Section ???):

$$t^2 + 2\alpha t - 1 = 0. \tag{1.1}$$

Without restricting the generality of the argumentation, we shall assume that $\alpha$ is positive. The problem is to find the root of Equation 1.1 having the smallest absolute value. You, reader, should have the answer somewhere in one corner of your brain, left over from high school mathematics:

$$t_{\min} = \sqrt{\alpha^1 + 1} - \alpha. \tag{1.2}$$

Let us now assume that $\alpha$ is very large, so large that adding 1 to $\alpha^2$ cannot be noticed within the machine precision. Then, the smallest of the solutions of Equation 1.1 becomes $t_{\min} \approx 0$, which is, of course, not true: the left-hand side of Equation 1.1 evaluates to $-1$.

Let us now rewrite Equation 1.1 for the variable $x = 1/t$. This gives the following equation:

$$x^2 - 2\alpha x - 1 = 0. \tag{1.3}$$

The smallest of the two solutions of Equation 1.1 is the largest of the two solutions of Equation 1.3. That is,

$$t_{\min} = \frac{1}{x_{\max}} = \frac{1}{\sqrt{\alpha^2 + 1} + \alpha}. \tag{1.4}$$

Now we have for large $\alpha$

$$t_{\min} \approx \frac{1}{2\alpha}. \tag{1.5}$$

This solution has certainly some rounding errors, but much fewer than the solution of Equation 1.2: the left-hand side of Equation 1.1 evaluates to $\frac{1}{4\alpha^2}$, which goes toward 0 for large $\alpha$, as it should be.

### 1.3.5 Wisdom From the Past

To close the subject of rounding errors, I would like to give the reader a different perspective. There is a big difference between exercising full control over rounding errors and giving a result with high precision. Granted, high-precision computation is required to minimize rounding errors. On the other hand, one only needs to keep the rounding errors under control to a level up to the precision required for the final results. There is no need to determine a result with nonsensical precision.

To illustrate the point, I am going to use a very old mathematical problem: the determination of the number $\pi$. The story is taken from the excellent book of Jan Gullberg, *Mathematics from the Birth of the Numbers* [Gullberg].

Around 300 BC, Archimedes devised a simple algorithm to approximate $\pi$. For a circle of diameter $d$, one computes the perimeter $p_{\text{in}}$ of a $n$-sided regular polygon inscribed within the circle and the perimeter $p_{\text{out}}$ of a $n$-sided regular polygon whose sides the tangent to the same circle. We have

$$\frac{p_{\text{in}}}{d} < \pi < \frac{p_{\text{out}}}{d}. \tag{1.6}$$

By increasing $n$, one can improve the precision of the determination of $\pi$. During antiquity and the Middle Ages, the computation of the perimeters was a formidable task, and an informal competition took place to find who could find the most precise approximation of the number $\pi$. In 1424, Jamshid Masud al-Kashi, a Persian scientist, published an approximation of $\pi$ with 16 decimal digits. The number of sides of the polygons was $3 \times 2^8$. This was quite an achievement, the last of its kind. After that, mathematicians discovered other means of expressing the number $\pi$.

In my eyes, however, Jamshid Masud al-Kashi deserves fame and admiration for the note added to his publication that places his result in perspective. He noted that the precision of his determination of the number $\pi$ was such that "the error in computing the perimeter of a circle with a radius 600,000 times that of earth would be less than the thickness of a horse's hair." The reader should know that the thickness of a horse's hair was a legal unit of measure in ancient Persia corresponding to roughly 0.7 mm. Using present-day knowledge of astronomy, the radius of the circle corresponding to the error quoted by al-Kashi is 147 times the distance between the sun and the Earth, or about three times the radius of the orbit of Pluto, the most distant planet of the solar system.

As Jan Gullberg notes in his book, "al-Kashi evidently had a good understanding of the meaninglessness of long chains of decimals." When dealing with numerical precision, you should ask yourself the following question: "Do I really need to know the length of Pluto's orbit to a third of the thickness of a horse's hair?"

## 1.4 Finding the Numerical Precision of a Computer

Object-oriented languages such as Smalltalk and Java give the opportunity to develop an application on one hardware platform and to deploy the application on other platforms running on different operating systems and hardware. It is a well-known fact that the marketing about Java was centered on the concept of "write once, run anywhere." What fewer people know is that this concept already existed for Smalltalk 10 years before the advent of Java.

Some numerical algorithms are carried until the estimated precision of the result becomes smaller than a given value, called the *desired precision*. Since an application can be executing on different hardware, the desired precision is best determined at run time.

The book of Press et al. [Press *et al.*] shows a clever code determining all the parameters of the floating-point representation of a particular computer. In this

book, we shall concentrate only on the parameters that are relevant for numerical computations. These parameters correspond to the instance variables of the object responsible to compute them. They are the following:

`radix` the radix of the floating-point representation—that is, $r$

`machinePrecision` the largest positive number that, when added to 1 yields 1

`negativeMachinePrecision` the largest positive number that, when subtracted from 1 yields 1

`smallestNumber` the smallest positive number different from 0

`largestNumber` the largest positive number that can be represented in the machine

`largestNumber` the largest positive number that can be represented in the machine

`defaultNumericalPrecision` the relative precision, that can be expected for a general numerical computation

`smallNumber` a number, that can be added to some value without noticeably changing the result of the computation

Computing the radix $r$ is done in two steps. First, one computes a number equivalent of the machine precision (see the next paragraph) assuming the radix is 2. Then, one keeps adding 1 to this number until the result changes. The number of added 1s is the radix.

The machine precision is computed by finding the largest integer $n$ such that

$$(1 + r^{-n}) - 1 \neq 0. \tag{1.7}$$

This is done with a loop over $n$. The quantity $\epsilon_+ = r^{-(n+1)}$ is the machine precision.

The negative machine precision is computed by finding the largest integer $n$ such that

$$\left(1 - r^{-n}\right) - 1 \neq 0. \tag{1.8}$$

Computation is made as for the machine precision. The quantity $\epsilon_- = r^{-(n+1)}$ is the negative machine precision. If the floating-point representation uses two-complement to represent negative numbers, the machine precision is larger than the negative machine precision.

To compute the smallest and largest number, one first computes a number whose mantissa is full. Such a number is obtained by building the expression $f = 1 - r \times \epsilon_-$. The smallest number is then computed by repeatedly dividing this value by the radix until the result produces an underflow. The last value obtained before an underflow occurs is the smallest number. Similarly, the largest number is computed by repeatedly multiplying the value $f$ until an overflow occurs. The last value obtained before an overflow occurs is the largest number.

The variable `defaultNumericalPrecision` contains an estimate of the precision expected for a general numerical computation. For example, one should consider that two numbers $a$ and $b$ are equal if the relative difference between them is less than the default numerical machine precision. This value of the default numerical machine precision has been defined as the square root of the machine precision.

The variable `smallNumber` contains a value, that can be added to some number without noticeably changing the result of the computation. In general, an expression of the type $\frac{0}{0}$ is undefined. In some particular case, however, one can define a value based on a limit. For example, the expression $\frac{\sin x}{x}$ is equal to 1 for $x = 0$. For algorithms, where such an undefined expression can occur, (Of course, after making sure that the ratio is well defined numerically), adding a small number to the numerator and the denominator can avoid the division by zero exception and can obtain the correct value. This value of the small number has been defined as the square root of the smallest number that can be represented on the machine.

### 1.4.1 Computer Numerical Precision—General Implementation

The computation of the parameters only needs to be executed once. We have introduced a specific class to hold the variables described earlier and made them available to any object.

Each parameter is computed using lazy initialization within the method bearing the same name as the parameter. Lazy initialization is used while all parameters may not be needed at a given time. Methods in charge of computing the parameters are all prefixed with the word *compute*.

### 1.4.2 Computer Numerical Precision—Smalltalk Implementation

Listing 1.1 shows the class `DhbFloatingPointMachine` responsible for computing the parameters of the floating-point representation. This class is implemented as a singleton class because the parameters need to be computed once only. For that reason, no code optimization was made, and priority is given to readability.

The computation of the smallest and largest numbers uses exceptions[9] to detect the underflow and the overflow. The method `showParameters` can be used to print the values of the parameters onto the Transcript window.

**Listing 1.1 Smalltalk code to find the machine precision**

| | |
|---|---|
| *Class* | `DhbFloatingPointMachine` |
| *Subclass of* | `Object` |

9. The code is using the implementation of Visual Age For Smalltalk™.

*Instance variable names:* `defaultNumericalPrecision radix machinePrecision negativeMachinePrecision smallestNumber largestNumber smallNumber largestExponentArgument`

*Class variable names:* `UniqueInstance`

### *Class Methods*

**new**

```
UniqueInstance = nil
    ifTrue: [ UniqueInstance := super new].
^UniqueInstance
```

**reset**

```
UniqueInstance := nil.
```

### *Instance Methods*

**computeLargestNumber**

```
| zero one floatingRadix fullMantissaNumber |
zero := 0 asFloat.
one := 1 asFloat.
floatingRadix := self radix asFloat.
fullMantissaNumber := one - ( floatingRadix * self
                                        negativeMachinePrecision).
largestNumber := fullMantissaNumber.
[ [ fullMantissaNumber := fullMantissaNumber * floatingRadix.
    largestNumber := fullMantissaNumber.
    true] whileTrue: [ ].
    ] when: ExAll do: [ :signal | signal exitWith: nil].
```

**computeMachinePrecision**

```
| one zero a b inverseRadix tmp x |
one := 1 asFloat.
zero := 0 asFloat.
inverseRadix := one / self radix asFloat.
machinePrecision := one.
[ tmp := one + machinePrecision.
  tmp - one = zero]
    whileFalse:[  machinePrecision := machinePrecision *
                                                inverseRadix].
```

**computeNegativeMachinePrecision**

```
| one zero floatingRadix inverseRadix tmp |
one := 1 asFloat.
```

```
zero := 0 asFloat.
floatingRadix := self radix asFloat.
inverseRadix := one / floatingRadix.
negativeMachinePrecision := one.
[ tmp := one - negativeMachinePrecision.
  tmp - one = zero]
    whileFalse:[ negativeMachinePrecision :=
                        negativeMachinePrecision * inverseRadix].
```

### computeRadix

```
| one zero a b tmp1 tmp2|
one := 1 asFloat.
zero := 0 asFloat.
a := one.
[ a := a + a.
  tmp1 := a + one.
  tmp2 := tmp1 - a.
  tmp2 - one = zero] whileTrue:[].
b := one.
[ b := b + b.
  tmp1 := a + b.
  radix := ( tmp1 - a) truncated.
  radix = 0 ] whileTrue: [].
```

### computeSmallestNumber

```
| zero one floatingRadix inverseRadix fullMantissaNumber |
zero := 0 asFloat.
one := 1 asFloat.
floatingRadix := self radix asFloat.
inverseRadix := one / floatingRadix.
fullMantissaNumber := one - ( floatingRadix * self
                                        negativeMachinePrecision).
smallestNumber := fullMantissaNumber.
[ [ fullMantissaNumber := fullMantissaNumber * inverseRadix.
    smallestNumber := fullMantissaNumber.
    true] whileTrue: [ ].
    ] when: ExAll do: [ :signal | signal exitWith: nil].
```

### defaultNumericalPrecision

```
defaultNumericalPrecision isNil
    ifTrue: [ defaultNumericalPrecision := self machinePrecision
                                                            sqrt].
^defaultNumericalPrecision
```

### largestExponentArgument

```
largestExponentArgument isNil
    ifTrue: [ largestExponentArgument := self largestNumber ln].
^largestExponentArgument
```

### largestNumber

```
largestNumber isNil
    ifTrue: [ self computeLargestNumber].
^largestNumber
```

### machinePrecision

```
machinePrecision isNil
    ifTrue: [ self computeMachinePrecision].
^machinePrecision
```

### negativeMachinePrecision

```
negativeMachinePrecision isNil
    ifTrue: [ self computeNegativeMachinePrecision].
^negativeMachinePrecision
```

### radix

```
radix isNil
    ifTrue: [ self computeRadix].
^radix
```

### showParameters

```
Transcript cr; cr;
        nextPutAll: 'Floating-point machine parameters'; cr;
        nextPutAll: '--------------------------------';cr;
        nextPutAll: 'Radix: '.
self radix printOn: Transcript.
Transcript cr; nextPutAll: 'Machine precision: '.
self machinePrecision printOn: Transcript.
Transcript cr; nextPutAll: 'Negative machine precision: '.
self negativeMachinePrecision printOn: Transcript.
Transcript cr; nextPutAll: 'Smallest number: '.
self smallestNumber printOn: Transcript.
Transcript cr; nextPutAll: 'Largest number: '.
self largestNumber printOn: Transcript.
```

### smallestNumber

```
smallestNumber isNil
```

```
    ifTrue: [ self computeSmallestNumber].
^smallestNumber
```

**smallNumber**

```
smallNumber isNil
    ifTrue: [ smallNumber := self smallestNumber sqrt].
^smallNumber
```

### 1.4.3 Computer Numerical Precision—Java Implementation

Listing 1.2 shows the Java implementation, which uses static variables and static methods. These methods and variables have been implemented in the class `DhbMath`. This class is responsible for computing the parameters of the floating-point representation, but it is also used to implement other mathematical utilities that are too simple to require the creation of a new class. The parameters of the floating-point representation need to be computed once only. Therefore, no code optimization was made; priority is given to readability.

Each parameter is retrieved with a method whose name is constructed with the standard prefix *get* followed with the name of the parameter. Methods in charge of computing the parameters are all prefixed with the word *compute*. The Java virtual machine does not raise an exception when floating underflow occurs. Thus, when computing the smallest number, underflow is simply detected by testing that the result is equal to 0. When computing the largest number, the Java virtual machine traps overflows to change the result into infinity. Thus, the overflow is detected when the result becomes infinite.

The method `printParameters` can be used to print the values of the parameters to any output stream. For example, the following code prints the parameters onto the standard output of the console.

```
DhbFloatingPointMachine.printParameters( System.out)
```

**Listing 1.2 Java code to find the machine precision**

```
package DhbFunctionEvaluation;

import java.io.PrintStream;

// This class implements additional mathematical functions
// and determines the parameters of the floating point representation.

// @author Didier H. Besset
```

```
public final class DhbMath
{
    // Typical meaningful precision for numerical calculations.
    static private double defaultNumericalPrecision = 0;
    // Typical meaningful small number for numerical calculations.
    static private double smallNumber = 0;
    // Radix used by floating-point numbers.
    static private int radix = 0;
    // Largest positive value that, when added to 1.0 yields 0.
    static private double machinePrecision = 0;
    // Largest positive value that, when subtracted from 1.0 yields 0.
    static private double negativeMachinePrecision = 0;
    // Smallest number different from zero.
    static private double smallestNumber = 0;
    // Largest possible number
    static private double largestNumber = 0;
    // Largest argument for the exponential
    static private double largestExponentialArgument = 0;
    // Values used to compute human readable scales.
    private static final double scales[] = {1.25, 2, 2.5, 4, 5, 7.5,
                                            8, 10};
    private static final double semiIntegerScales[] = {2, 2.5, 4, 5,
                                                       7.5, 8, 10};
    private static final double integerScales[] = {2, 4, 5, 8, 10};

private static void computeLargestNumber()
{
    double floatingRadix = getRadix();
```

```
        double fullMantissaNumber = 1.0d -
                            floatingRadix * getNegativeMachinePrecision();
        while ( !Double.isInfinite( fullMantissaNumber) )
        {
            largestNumber = fullMantissaNumber;
            fullMantissaNumber *= floatingRadix;
        }
    }
    private static void computeMachinePrecision()
    {
        double floatingRadix = getRadix();
        double inverseRadix = 1.0d / floatingRadix;
        machinePrecision = 1.0d;
        double tmp = 1.0d + machinePrecision;
        while ( tmp - 1.0d != 0.0d)
        {
            machinePrecision *= inverseRadix;
            tmp = 1.0d + machinePrecision;
        }
    }
    private static void computeNegativeMachinePrecision()
    {
        double floatingRadix = getRadix();
        double inverseRadix = 1.0d / floatingRadix;
        negativeMachinePrecision = 1.0d;
        double tmp = 1.0d - negativeMachinePrecision;
        while ( tmp - 1.0d != 0.0d)
        {
            negativeMachinePrecision *= inverseRadix;
            tmp = 1.0d - negativeMachinePrecision;
        }
    }
    private static void computeRadix()
    {
        double a = 1.0d;
        double tmp1, tmp2;
        do { a += a;
             tmp1 = a + 1.0d;
             tmp2 = tmp1 - a;
            } while ( tmp2 - 1.0d != 0.0d);
        double b = 1.0d;
        while ( radix == 0)
        {
            b += b;
            tmp1 = a + b;
            radix = (int) ( tmp1 - a);
```

```
    }
}
private static void computeSmallestNumber()
{
    double floatingRadix = getRadix();
    double inverseRadix = 1.0d / floatingRadix;
    double fullMantissaNumber = 1.0d - floatingRadix *
getNegativeMachinePrecision();
    while ( fullMantissaNumber != 0.0d )
    {
        smallestNumber = fullMantissaNumber;
        fullMantissaNumber *= inverseRadix;
    }
}
public static double defaultNumericalPrecision()
{
    if ( defaultNumericalPrecision == 0 )
        defaultNumericalPrecision = Math.sqrt( getMachinePrecision());
    return defaultNumericalPrecision;
}

// @return boolean    true if the difference between a and b is
// less than the default numerical precision
// @param a double
// @param b double

public static boolean equal( double a, double b)
{
    return equal( a, b, defaultNumericalPrecision());
}

// @return boolean    true if the relative difference between a and b
// is less than precision
// @param a double
// @param b double
// @param precision double

public static boolean equal( double a, double b, double precision)
{
    double norm = Math.max( Math.abs(a), Math.abs( b));
    return norm < precision || Math.abs( a - b) < precision * norm;
}
public static double getLargestExponentialArgument()
{
    if ( largestExponentialArgument == 0 )
        largestExponentialArgument = Math.log(getLargestNumber());
```

```
    return largestExponentialArgument;
}
// (c) Copyrights Didier BESSET, 1999, all rights reserved.

public static double getLargestNumber()
{
    if ( largestNumber == 0 )
        computeLargestNumber();
    return largestNumber;
}
public static double getMachinePrecision()
{
    if ( machinePrecision == 0 )
        computeMachinePrecision();
    return machinePrecision;
}
public static double getNegativeMachinePrecision()
{
    if ( negativeMachinePrecision == 0 )
        computeNegativeMachinePrecision();
    return negativeMachinePrecision;
}
public static int getRadix()
{
    if ( radix == 0 )
        computeRadix();
    return radix;
}
public static double getSmallestNumber()
{
    if ( smallestNumber == 0 )
        computeSmallestNumber();
    return smallestNumber;
}
public static void printParameters( PrintStream printStream)
{
    printStream.println( "Floating-point machine parameters");
    printStream.println( "--------------------------------");
    printStream.println( " ");
    printStream.println( "radix = "+ getRadix());
    printStream.println( "Machine precision = "
                                          + getMachinePrecision());
    printStream.println( "Negative machine precision = "
                                      + getNegativeMachinePrecision());
    printStream.println( "Smallest number = "+ getSmallestNumber());
    printStream.println( "Largest number = "+ getLargestNumber());
```

```
    return;
}
public static void reset()
{
    defaultNumericalPrecision = 0;
    smallNumber = 0;
    radix = 0;
    machinePrecision = 0;
    negativeMachinePrecision = 0;
    smallestNumber = 0;
    largestNumber = 0;
}

// This method returns the specified value rounded to
// the nearest integer multiple of the specified scale.

// @param value number to be rounded
// @param scale defining the rounding scale
// @return rounded value

public static double roundTo(  double value, double scale)
{
    return Math.round( value / scale) * scale;
}

    // Round the specified value upward to the next scale value.
    // @param the value to be rounded.
    // @param a fag specified whether integer scale are used, otherwise double scale
       is used.
    // @return a number rounded upward to the next scale value.

    public static double roundToScale( double value, boolean integerValued)
    {
        double[] scaleValues;
        int orderOfMagnitude = (int) Math.floor( Math.log( value)
                               / Math.log( 10.0));
        if ( integerValued )
        {
            orderOfMagnitude = Math.max( 1, orderOfMagnitude);
            if ( orderOfMagnitude == 1)
                scaleValues = integerScales;
            else if ( orderOfMagnitude == 2)
                scaleValues = semiIntegerScales;
            else
                scaleValues = scales;
        }
```

```
            else
                scaleValues = scales;
            double exponent = Math.pow( 10.0, orderOfMagnitude);
            double rValue = value / exponent;
            for ( int n = 0; n < scaleValues.length; n++)
            {
                if ( rValue <= scaleValues[n])
                    return scaleValues[n] * exponent;
            }
            return exponent;    // Should never reach here
        }

    // (c) Copyrights Didier BESSET, 1999, all rights reserved.
    // @return double

    public static double smallNumber()
    {
        if ( smallNumber == 0 )
            smallNumber = Math.sqrt( getSmallestNumber());
        return smallNumber;
    }
    }
```

## 1.5 Comparing Floating-Point Numbers

It is very surprising to see how frequently questions arise about the lack of equality between two floating-point numbers in the Smalltalk and Java electronic discussion groups. As we have seen in Section 1.3.2, one should always expect the result of two different computations that should have yielded the same number from a mathematical standpoint to be different using a finite numerical representation. Somehow the computer courses are not giving enough emphasis about floating-point numbers.

> So, how should you check the equality of two floating-point numbers? The practical answer is "Thou shalt not!"

As you will see, the algorithms in this book only compare numbers but never check for equality. If you cannot escape the need for a test of equality, however, the best solution is to create methods to do this. Since the floating-point representation is keeping a constant relative precision, comparison must be made using relative error. Let $a$ and $b$ be the two numbers to be compared. One should build the following expression:

$$\epsilon = \frac{|a-b|}{\max\left(|a|,|b|\right)}. \tag{1.9}$$

The two numbers can be considered equal if $\epsilon$ is smaller than a given number $\epsilon_{max}$. If the denominator of the fraction in Equation 1.9 is less than $\epsilon_{max}$, than the two numbers can be considered equal. For lack of information on how the numbers *a* and *b* have been obtained, one uses for $\epsilon_{max}$ the default numerical precision defined in Section 1.4. If one can determine the precision of each number, then the method `relativelyEqual` can be used.

### 1.5.1 Comparing Floating-Point Numbers—Smalltalk Code

In Smalltalk this means adding a new method to the class Number as shown in Listing 1.3.

**Listing 1.3 Comparison of floating-point numbers in Smalltalk**

*Class* `Number`

*Subclass of* `Magnitude`

***Instance Methods***

**equalsTo:** `aNumber`

```
^self relativelyEqualsTo: aNumber upTo: DhbFloatingPointMachine
                                        new defaultNumericalPrecision
```

**relativelyEqualsTo:** `aNumber` **upTo:** `aSmallNumber`

```
| norm |
norm := self abs max: aNumber abs.
^norm <= DhbFloatingPointMachine new defaultNumericalPrecision
    or: [ (self - aNumber) abs < ( aSmallNumber * norm)]
```

## 1.6 Speed Consideration

Some people may think that implementing numerical methods for object-oriented languages such as Smalltalk or Java is just a waste of time. Those languages are notoriously slow—or so they think.

First, things should be put in perspective with other actions performed by the computer. If a computation does not take longer than the time needed to refresh a screen, it does not really matter when the application is interactive. For example, performing a least-square fit to a histogram in Smalltalk and Java and drawing the resulting fitted function is usually hardly perceptible to the eye on a personal computer using a 200MHz Pentium. Thus, even though a C version runs 10 times faster, it does not make any difference for the end user. The main difference comes, however, when you need to modify the code. Object-oriented software is well known

**TABLE 1.1** Compared execution speed among C, Smalltalk, and Java

| Operation | Units | C | Smalltalk | Java |
|---|---|---|---|---|
| Polynomial $10^{th}$ degree | msec. | 1.1 | 27.7 | 9.0 |
| Neville interpolation (20 points) | msec. | 0.9 | 11.0 | 0.8 |
| LUP matrix inversion ($100 \times 100$) | sec. | 3.9 | 22.9 | 1.0 |

for its maintainability. As 80% of the code development is spent on maintenance, this aspect should be considered first.

Table 1.1 shows measured speed of execution for some of the algorithms exposed in this book. Timing was done on a personal computer equipped with a Pentium II clocked at 200MHz and running Windows NT workstation 4.0. The C code used is the code of [Press *et al.*] compiled with the C compiler Visual C$^{++}$ 4.0 from Microsoft Corporation. The time needed to allocate memory for intermediate results was included in the measurement of the C code; otherwise the comparison with object-oriented code would not be fair. The Smalltalk code was run under version 4.0 of Visual Age™for Smalltalk from IBM Corporation using the ENVY benchmark tool provided. The Java code was run under version 2.0 of Visual Age™for Java from IBM Corporation. Elapsed times were measured by repeating the measured evaluation a sufficient number of times so that the error caused by the CPU clock is less that the last digit shown in the final result.

One can see that object-oriented code is quite efficient, especially when it comes to complex algorithms: good object-oriented code can actually beat up C code.

My early tests with Java a couple of years ago showed that Java was 5 to 10 times slower than C. One can see that vendors did a great job in optimizing the generated code and in accelerating the virtual machine. I would like to see the same efforts going in optimizing Smalltalk. The spectacular improvement of Java shows that it is possible. Actually, my early tests made with Visual Smalltalk™from Digitalk Inc.[10] were five times better.

Admittedly, I would not use Smalltalk today to build a structural analysis program, but Java would certainly be a contender. Nevertheless, I have successfully built data-mining Smalltalk applications using all the code[11] presented in this book. These applications were not perceived as slow by the end user since most of the computer time was spent drawing the data.

10. Unfortunately, the future of Visual Smalltalk, now owned by Cincom Inc., is quite uncertain at the time of writing.

11. I want to emphasize here that all the code in this book is real code, that I have used personally in real applications.

### 1.6.1 Smalltalk Particular

Smalltalk has an interesting property: a division between two integers is by default kept as a fraction. This prevents rounding errors. For example, the multiplication of a matrix of integer numbers with its inverse always yields an exact identity matrix (see Section ??? for definitions of these terms).

There is, however, a price to pay for the perfection offered by fractions. When fractions are used, the computing time often becomes prohibitive. Resulting fractions are often composed of large integers, which slows down the computing. In the case of matrix inversion, this results in an increase in computing time by several orders of magnitude.

For example, one of my customers was inverting a $301 \times 301$ matrix with the code of Section ???. The numbers used to build the matrix were obtained from a measuring device (using an ADC) and were thus integers. The inversion time was over 2 hours. (This particular customer was a *very* patient person!) After converting the matrix components to floating numbers, the inversion time became less than 30 seconds!

If you are especially unlucky, you may run out of memory when attempting to store a particularly long integer. Thus, it is always a good idea to use floating[12] numbers instead of fractions unless absolute accuracy is your primary goal. My experience has been that using floating numbers speeds up the computation by at least an order of magnitude. In the case of complex computations such as matrix inversion or least-square fit, this approach can become prohibitive.

## 1.7 Conventions

Equations presented in this book are using standard international mathematical notation as described in [Knudth 1]. Each section is trying to made a quick derivation of the concepts needed to fully understand the mathematics behind the scene. For readers in a hurry, the equations used by the implementation are flagged as the following sample equation:

$$\ln ab = \ln a + \ln b. \tag{1.10}$$

When such an equation is encountered, the reader is sure that the expression is implemented in the code.

In general, the code presented in this book adheres to conventions widely used in each language. Having said that, we have departed from the widely used conventions in a few instances.

12. In most available Smalltalk versions, the class Float corresponds to floating numbers with double precision. VisualWorks makes the difference between `Float` and `Double`

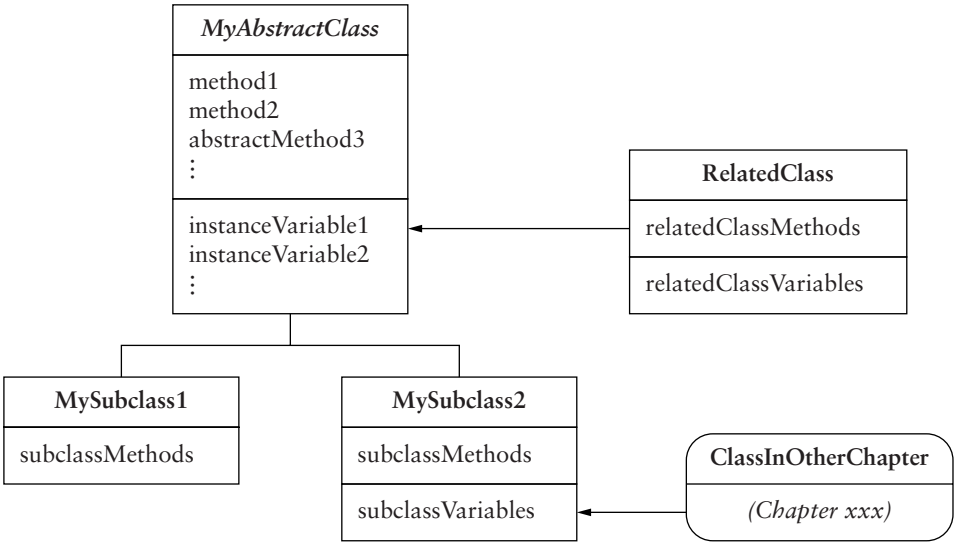


**FIG. 1.2** A typical class diagram

### 1.7.1 Class Diagrams

When appropriate, a class diagram is shown at the beginning of each chapter. This diagram shows the hierarchy of the classes described in the chapter and eventually the relations with classes of other chapters. The diagrams are drawn using the conventions of the book on design patterns [Gamma *et al.*].

Figure 1.2 shows a typical class diagram. A rectangular box with two or three parts represents a class. The top part contains the name of the class in boldface. If the class is an abstract class, the name in shown in italicized boldface. In Figure 1.2, the classes `RelatedClass`, `MySubClass1`, and `MySubclass2` are concrete classes; `MyAbstractClass` is an abstract class. The second part of the class box contains a list of the public instance methods. The name of an abstract method is written in italics—for example, `abstractMethod3` in the class `MyAbstractClass` of Figure 1.2. The third part of the class box, if any, contains the list of all instance variables. If the class does not have any instance variable, the class box consists of only two parts—for example, the class `MySubClass1` of Figure 1.2.

A vertical line with a triangle indicates class inheritance. If there are several subclasses, the line branches at the triangle, as this is the case in Figure 1.2. A horizontal arrow beginning with a diamond (the UML aggregation symbol) indicates the class of an instance variable. For example, Figure 1.2 indicates that the instance variable `instanceVariable2` of the class `MyAbstractClass` must be an instance of the class `RelatedClass`. The diamond is black if the instance variable is a collection of instances of the class. A class within a rectangle with rounded corners represents a class already discussed in an earlier chapter; the reference to the chapter is written

below the class name. Class `ClassInOtherChapter` in Figure 1.2 is such a class. To save space, we have used the Java class names and the Smalltalk method names. It is quite easy to identify methods needing parameters when one uses Smalltalk method names: a semicolon in the middle or at the end of the method name indicates a parameter. Please refer to Appendix ??? for more details on Smalltalk methods.

### 1.7.2 Smalltalk Code

Most of the Smalltalk systems do not support name spaces. As a consequence, it has becomed a convention to prefix all class names with a three-letter code identifying the origin of the code. In this book, the names of the Smalltalk classes are all prefixed with my initials.

There are several ways to store constants needed by all instances of a class. One way is to store the constants in class variables. This requires each class to implement an initialization method, that sets the desired values into the class variables. Another way is to store the constants in a pool dictionary. Here, also, an initialization method is required. In my opinion, pool dictionaries are best used for texts, as they provide a convenient way to change all text from one language to another. Sometimes the creation of a singleton object is used. This approach is especially useful when the constants are installation-specific and, therefore, must be determined at the beginning of the application's execution, such as the precision of the machine (see Section 1.4). Finally constants that are not likely to change can be stored in the code, either `technique` or `way of doing`. This is acceptable as long as it is done at a unique place. In this book, most constants are defined in class methods.

By default, a Smalltalk method returns `self`. For initialization methods, however, we write this return explicitly (`^self`) to ease reading. This adheres to the intention revealing patterns of Kent Beck [Beck].

In [Alpert *et al.*], it is recommended to use the method name `default` to implement a singleton class. In this book, this convention is not followed. In Smalltalk, however, the normal instance creation method is `new`. Introducing a method `default` for singleton classes has the effect of departing from this more ancient convention. In fact, requiring the use of `default` amounts to revealing to the client the details of implementation used by the class. This is in clear contradiction with the principle of hiding the implementation to the external world. Thus, singleton classes in all code presented in this book are obtained by sending the method `new` to the class. A method named `default` is reserved for the very semantic of the word *default*: the instance returned by these methods is an instance initialized with some default contents, well specified. Whether or not the instance is a singleton is not the problem of the client application.

### 1.7.3 Java Code

Java forces code to be maintained as files. A file may contain several classes, but a class cannot be defined across several files. In this book, classes are always presented

in a single listing. A note placed at the end of each listing mentions eventual methods discussed in a subsequent sections.

Java supports name space for class names. The unit of name space is called a *package*. The names of the packages containing the classes described in this book are all prefixed with my initials, as recommended by the Java designers. The class names are not prefixed.

The designers of the Java language have chosen to conform to the C syntax as much as possible. In particular, pre- and postincrement operators are available. In the code presented in this book, the use of these operators is restricted to indices. If a variable is used for counting, the assignment with coupled operation is used instead of self incrementation. For example, incrementing a counter is written as

```
counter += 1;
```
(1.11)

whereas incrementing an index is written as

```
index++;
```
(1.12)

This adheres to the intention-revealing patterns of Beck [Beck]. As the two statements are strictly equivalent, the compiler is expected to generate the most efficient byte code.

## 1.8 Road Map

This last section of the introduction describes the road map of the algorithms discussed in the book chapter by chapter. Figure 1.3 shows a schematic view of the major classes discussed in this book together with their dependency relations. In this figure, abstract classes are represented with an ellipse, concrete classes with a rectangle. Dependencies between the classes are represented by lines going from one class to another; the dependent class is always located below. Chapters in which the classes are discussed are drawn as grayed rectangles with rounded corners. I hope the reader will not be scared by the complexity of the figure. Actually, the figure should be more complex, as the classes `Vector` and `Matrix` are used by most objects located in Chapters ??? and following. To preserve the readability of Figure 1.3, the dependency connections for these two classes have been left out.

Chapter 2 presents a general representation of mathematical functions. Examples are shown. A concrete implementation of polynomial is discussed. Finally, three library functions are given: the error function, the gamma function, and the beta function.

Chapter 3 discusses interpolation algorithms. A discussion explains when interpolation should be used and which algorithm is more appropriate to which data.

Chapter 4 presents a general framework for iterative process. It also discusses a specialization of the framework to iterative process with a single numerical result. This framework is widely used in the rest of the book.

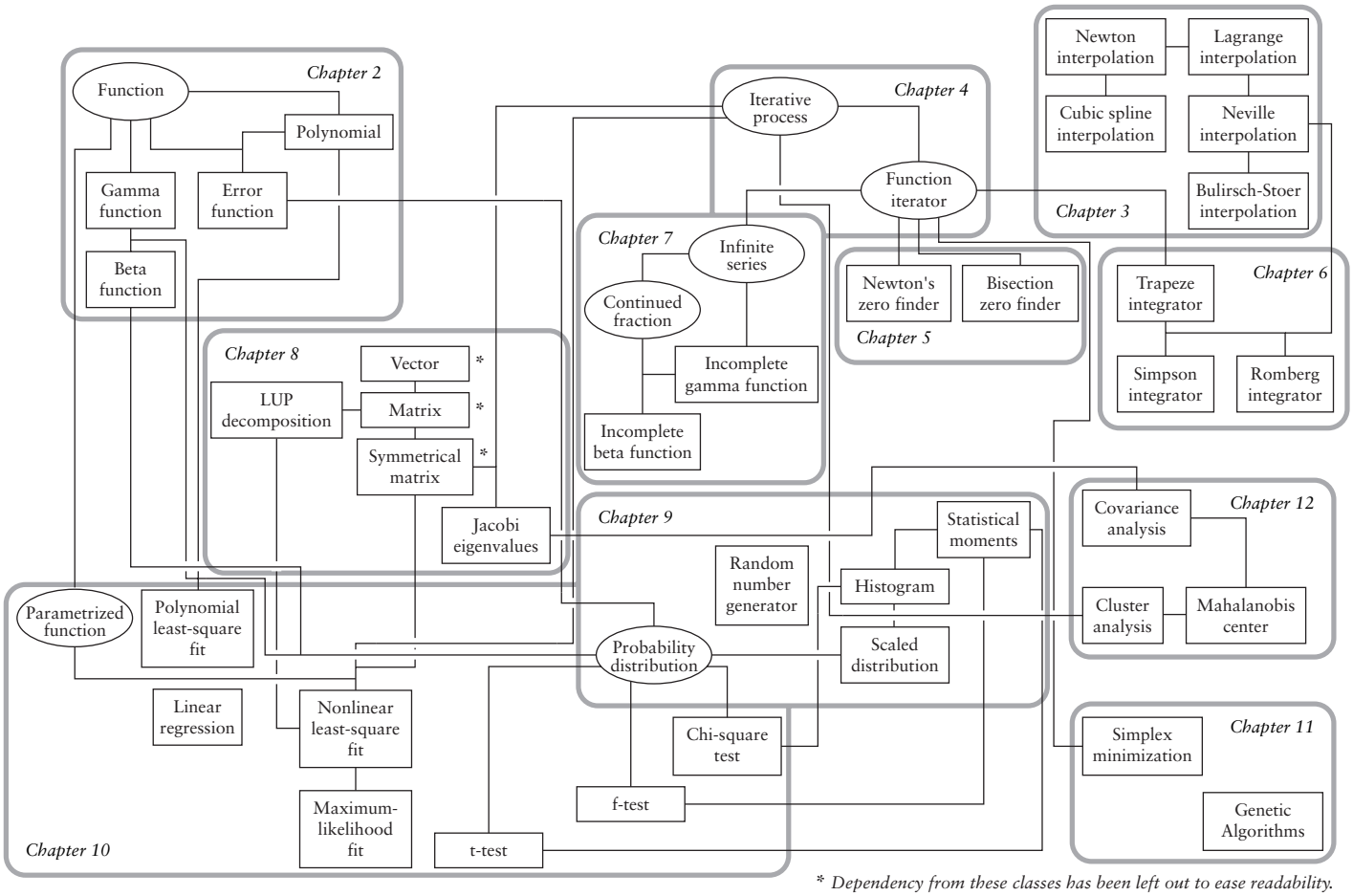


**FIG. 1.3** Book road map

Chapter 5 discusses two algorithms to find the zeroes of a function: bisection and Newton's zero finding algorithms. Both algorithms use the general framework of Chapter 4.

Chapter 6 discusses several algorithms to compute the integral of a function. All algorithms are based on the general framework of Chapter 4. This chapter also uses an interpolation technique from Chapter 3.

Chapter 7 discusses the specialization of the general framework of Chapter 4 to the computation of infinite series and continued fractions. The incomplete gamma function and incomplete beta function are used as concrete examples to illustrate the technique.

Chapter ??? presents a concrete implementation of vector and matrix algebra. It also discusses algorithms to solve systems of linear equations. Algorithms to compute matrix inversion and the finding of eigenvalues and eigenvectors are exposed. Elements of this chapter are used in other parts of this book.

Chapter ??? presents tools to perform statistical analysis. Random number generators are discussed. I give an abstract implementation of a probability distribution with a concrete example of the most important distributions. The implementation of other distributions is given in Appendix D. This chapter uses techniques from Chapters 2, 5, and 6.

Chapter ??? discusses the test of hypothesis and estimation, giving an implementation of the *t*- and *F*-tests. It presents a general framework to implement least-square fit and maximum-likelihood estimation. Concrete implementations of least-square fit for linear and polynomial dependence are given. A concrete implementation of the maximum-likelihood estimation is given to fit a probability distribution to a histogram. This chapter uses techniques from Chapters 2, 4, ???, and ???.

Chapter ??? discusses some techniques used to maximize or minimize a function: classical algorithms (simplex, hill climbing) as well as new ones (genetic algorithms). All these algorithms use the general framework for iterative process discussed in Chapter 4.

Chapter ??? discusses the modern data-mining techniques: correlation analysis, cluster analysis, and neural networks. A couple of methods that I invented are also discussed. This chapter uses, directly or indirectly, techniques from all chapters of this book.

CHAPTER 2

# Function Evaluation

*Qu'il n'y ait pas de réponse n'excuse pas l'absence de questions.*[1]

—Claude Roy

Many mathematical functions used in numerical computation are defined by an integral, a recurrence formula, or a series expansion. While such definitions can be useful to a mathematician, they are usually quite complicated to implement on a computer. For one, not every programmer knows how to evaluate an integral numerically. (The good news is that they will if they read the present book see Chap. 6). Then, there is the problem of accuracy. Finally, the evaluation of the function as defined mathematically is often too slow to be practical.

Before computers were heavily used, however, people had already found efficient ways of evaluating complicated functions. These methods are usually precise enough and extremely fast. This chapter exposes several functions that are important for statistical analysis. The *Handbook of Mathematical Functions* by Abramovitz and Stegun [Abramovitz & Stegun] contains a wealth of such function definitions and describes many ways of evaluating them numerically. Most approximations used in this chapter have been taken from this book.

This chapter opens with general considerations on how to implement the concept of function. Then, polynomials are discussed as an example of concrete function implementation. The rest of this chapter introduces three classical functions: the error function, the gamma function, and the beta function. We shall use this functions in Chapters ??? and ???. Because these functions are fundamental functions used in many areas of mathematics, they are implemented as library functions—such as a sine, log, or exponential—instead of using the general function formalism described in the first section.

1. The absence of answer does not justify the absence of question.

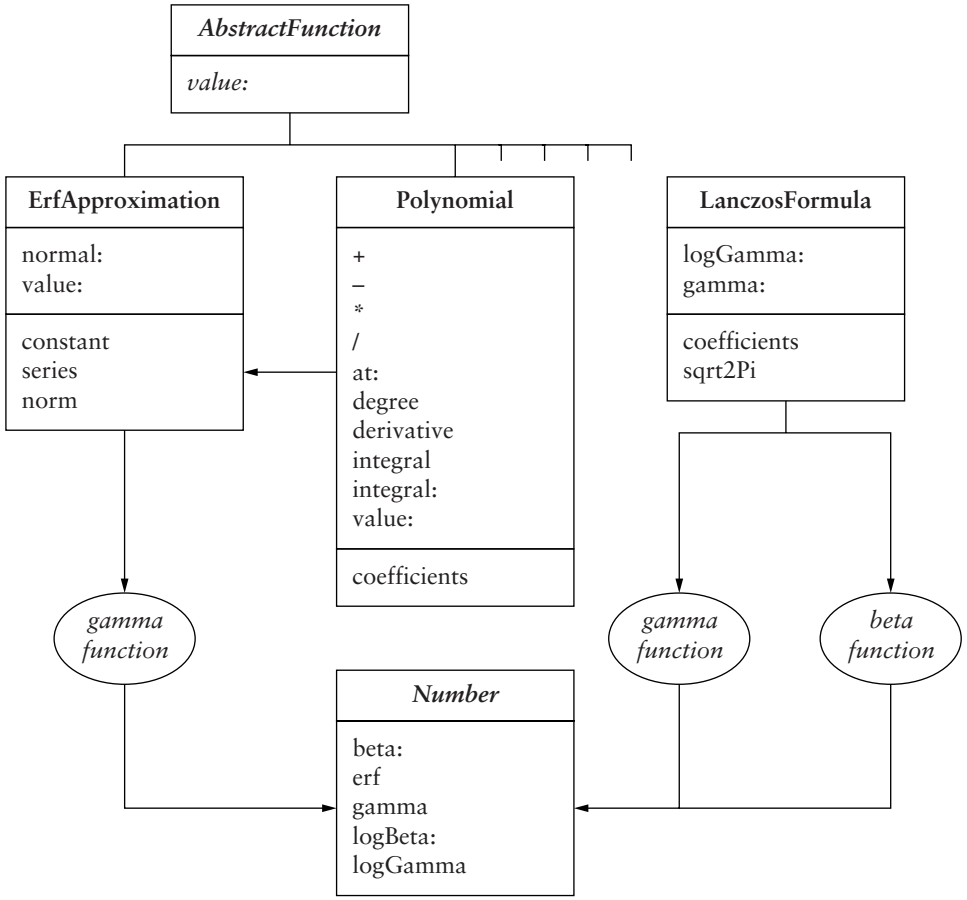


**FIG. 2.1** Smalltalk classes related to functions

The two language implementations are quite different because the concepts are implemented at a very low level. Figure 2.1 shows the diagram of the Smalltalk classes described in this chapter. Here I have used special notations to indicate that the functions are implemented as library functions. The functions are represented by ovals, and arrows shows which class is used to implement a function for the class `Number`.

Figure 2.2 shows the diagram of the Java classes described in this chapter. The strangeness of the box for the `NormalDistribution` class indicates that only the variables and methods relevant to this chapter are shown here. Both variables and methods are static.

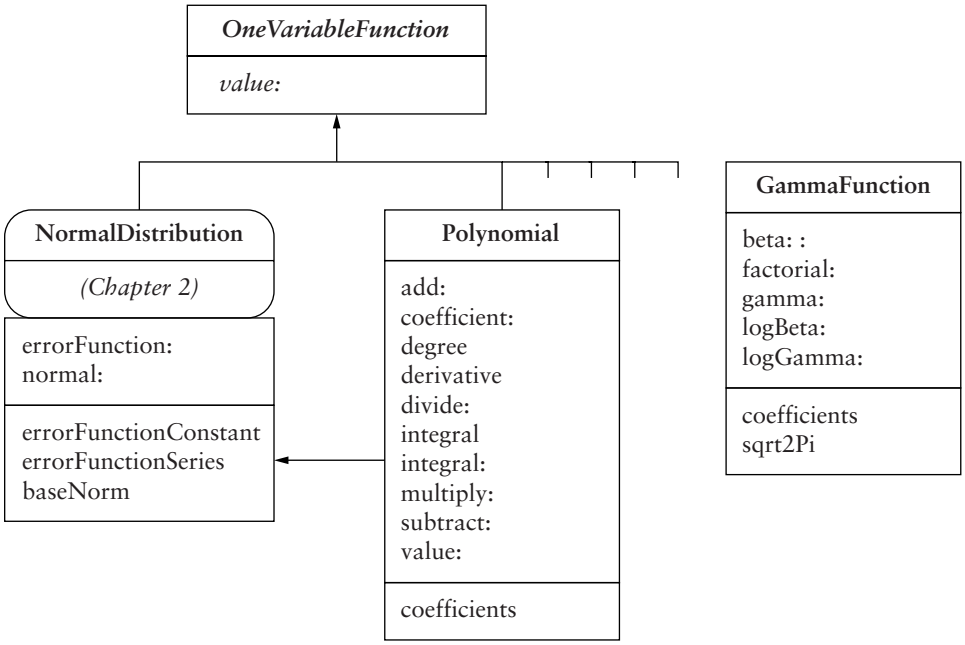


**FIG. 2.2** Java classes related to functions

## 2.1 Function Concept

A *mathematical function* is an object associating a value, to a variable. If the variable is a single value, one talks about a *one-variable function*. If the variable is an array of values, one talks about a *multivariable function*. Other types of variables are possible but will not be covered in this book.

I shall assume that the reader is familiar with elementary concepts about functions—namely, derivatives and integrals. I shall concentrate mostly on implementation issues.

### 2.1.1 Function—Smalltalk Implementation

A function in Smalltalk can be readily implemented with a block closure. Block closures in Smalltalk are treated like objects; thus, they can be manipulated as any other objects. For example, the one variable-function defined as

$$f(x) = \frac{1}{x}, \tag{2.1}$$

can be implemented in Smalltalk as

```
f := [:x | 1 / x]
```
(2.2)

Evaluation of a block closure is supplied by the method `value:`. For example, to compute the inverse of 3, one writes

```
f value: 3
```
(2.3)

If the function is more complex, a block closure may not be the best solution to implement a function. Instead, a class can be created with some instance variables to hold any constants and/or partial results. To be able to use functions indifferently implemented as block closures or as classes, one uses polymorphism. Each class implementing a function must implement a method `value:`. Thus, any object evaluating a function can send the same message selector—namely `value:`—to the variable holding the function.

To evaluate a multivariable function, the argument of the method `value:` is an array or a vector (see Section ???). Thus, in Smalltalk multivariable functions can follow the same polymorphism as for one-variable functions.

### 2.1.2 Function—Java Implementation

The situation in Java is a little more complex. Unlike Smalltalk, Java does not have a concept of block closure. Therefore, code cannot be manipulated as an object. In addition, strong typing results in our making different declarations for one-variable and multivariable functions.

One way to keep the same generality as in Smalltalk would be to use a general object declaration and use casting. However, this approach departs from the philosophy of a strongly typed language, which is central to the design of Java. It was therefore not considered.

Java, however, has introduced a very convenient concept, the interface. In this case, we need two of them: an interface to declare one-variable functions and one to declare multivariable functions. Thus, a class interacting with a function can declare the function—as an instance variable or a method argument—with the required interface.

Listing 2.1 shows the implementation of the Java interface `OneVariableFunction`.

---

**Listing 2.1 Java implementation of the interface `OneVariableFunction`**

```
package DhbInterfaces;

// OneVariableFunction is an interface for mathematical functions of
// a single variable, that is functions of the form f(x).

// @author Didier H. Besset
```

```
public interface OneVariableFunction
{

// Returns the value of the function for the specified variable value.

public double value( double x);
}
```

If the function is simple enough, it can be implemented as an inner class[2]. For example, the definition of the inverse function as an inner class is shown in Code Example 2.1.

**Code Example 2.1**

```
new OneVariableFunction() {
   public double value( double x)
   {return 1/x;}}
```

A more complex function must be implemented as a full-fledged class. A function class can be the subclass of any class as long as it implements the interface. Numerous examples of such function classes will appear in the rest of this book.

A further example of the use of the `OneVariableFunction` interface is given in Listing 2.2 showing the implementation of the Java class `FunctionDerivative` using the interface `OneVariableFunction`. This class allows the computation of the derivative of any function of one variable. The evaluation of the derivative uses the following mathematical approximation:

$$\frac{df(x)}{dx} = \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x-\epsilon)}{2\epsilon}. \tag{2.4}$$

This symmetrical definition of the derivative guarantees a higher accuracy than the usual one-sided definition. Of course, evaluating the derivative with this class is slower than coding the derivative explicitly as it requires two computations of the original function for every evaluation. But it is better than nothing, if the function whose derivative is needed is too complicated to be expressed analytically. Last but not least, this class can be used to check whether the supplied derivatives are indeed correct.

An example of use of the class `FunctionDerivative` is shown in Section 5.3.2.

**Listing 2.2 Java implementation of a generic derivative evaluation**

```
package DhbFunctionEvaluation;
```

2. Inner classes are a feature of Java 1.1 and higher versions.

```
import DhbInterfaces.OneVariableFunction;

// Evaluate an approximation of the derivative of a given function.

// @author Didier H. Besset

public final class FunctionDerivative implements OneVariableFunction
{

   // Function for which the derivative is computed.

   private OneVariableFunction f;

   // Relative interval variation to compute derivative.

   private double relativePrecision = 0.0001;

// Constructor method.
// @param func the function for which the derivative is computed.

public FunctionDerivative( OneVariableFunction func)
{
    this( func, 0.000001);
}

// Constructor method.
// @param func the function for which the derivative is computed.
// @param precision the relative step used to compute the derivative.

public FunctionDerivative( OneVariableFunction func, double precision)
{
   f = func;
   relativePrecision = precision;
}

// Returns the value of the function's derivative
// for the specified variable value.

public double value( double x)
{
   double x1 = x == 0 ? relativePrecision
                      : x * ( 1 + relativePrecision);
   double x2 = 2 * x - x1;
   return (f.value(x1) - f.value(x2)) / (x1 - x2);
}
}
```

## 2.2 Polynomials

Polynomials are quite important in numerical methods because they are often used in approximating functions. For example, Section 2.3 shows how the error function can be approximated with the product of normal distribution times a polynomial.

Polynomials are also useful in approximating functions, which are determined by experimental measurements in the absence of any theory on the nature of the function. For example, the output of a sensor detecting a coin is dependent on the temperature of the coin mechanism. This temperature dependence cannot be predicted theoretically because it is a difficult problem. Instead, one can measure the sensor output at various controlled temperatures. These measurements are used to determine the coefficients of a polynomial reproducing the measured temperature variations. The determination of the coefficients is performed using a polynomial least-square fit (see Section ???). Using this polynomial, the correction for a given temperature can be evaluated for any temperature within the measured range.

The implementations in both languages are discussed in great detail. The reader is advised to read carefully both implementation sections as many techniques are introduced at this occasion. Later on those techniques will be mentioned with no further explanations.

### 2.2.1 Mathematical Definitions

A *polynomial* is a special mathematical function whose value is computed as follows:

$$P(x) = \sum_{k=0}^{n} a_k x^k. \tag{2.5}$$

In the equation above, $n$ is called the *degree* of the polynomial. For example, the second order polynomial

$$x^2 - 3x + 2 \tag{2.6}$$

has degree 2. It represents a parabola crossing the $x$-axis at points 1 and 2 and having a minimum at $x = 2/3$. The value of the polynomial at the minimum is $-1/4$.

In equation 2.5 the numbers $a_0, \ldots a_n$ are called the *coefficients* of the polynomial. Thus, a polynomial can be represented by the array $\{a_0, \ldots a_n\}$. For example, the polynomial of equation 2.6 is represented by the array $\{1, -3, 2\}$.

Evaluating equation 2.5 as such is highly inefficient since one must raise the variable to an integral power at each term. The required number of multiplication is of the order of $n^2$. There is, of course, a better way to evaluate a polynomial. It

consists of factoring out $x$ before the evaluation of each term.[3] Equation 2.7 shows the resulting expression:

$$(x) = a_0 + x\left\{a_1 + x[a_2 + x(a_3 + \cdots)]\right\} \tag{2.7}$$

Evaluating this expression now requires only multiplications. The resulting algorithm is quite straightforward to implement. Equation 2.7 is called *Horner's rule* because it was first published by W. G. Horner in 1819. However, 150 years earlier, Isaac Newton was already using this method to evaluate polynomials.

In Section 5.3 we shall require the derivative of a function. For polynomials this is rather straightforward. The derivative is given by:

$$\frac{dP(x)}{dx} = \sum_{k=1}^{n} k a_k x^{k-1}. \tag{2.8}$$

Thus, the derivative of a polynomial with $n$ coefficients is another polynomial, with $n-1$ coefficients[4] derived from the coefficients of the original polynomial as follows:

$$a'_k = (k+1)a_{k+1} \quad \text{for } k = 0, \ldots, n-1. \tag{2.9}$$

For example, the derivative of 2.6 is $2x - 3$.

The integral of a polynomial is given by:

$$\int_0^x P(t)dt = \sum_{k=0}^{n} \frac{a_k}{k+1} x^{k+1}. \tag{2.10}$$

Thus, the integral of a polynomial with $n$ coefficients is another polynomial, with $n+1$ coefficients derived from the coefficients of the original polynomial as follows:

$$\bar{a}_k = \frac{a_{k-1}}{k} \quad \text{for } k = 1, \ldots, n+1. \tag{2.11}$$

For the integral, the coefficient $\bar{a}_0$ is arbitrary and represents the value of the integral at $x = 0$. For example, the integral of 2.6, which has the value $-2$ at $x = 0$, is the polynomial

$$\frac{x^3}{3} - \frac{3^2}{2} + 2x - 2. \tag{2.12}$$

Conventional arithmetic operations are also defined on polynomials and have the same properties[5] as for signed integers.

---

3. This is actually the first program I ever wrote in my first computer programming class. Back in 1969, the language in fashion was ALGOL.

4. Notice the change in the range of the summation index in equation 2.8.

5. The set of polynomials is a vector space in addition to being a ring.

Adding or subtracting two polynomials yields a polynomial whose degree is the maximum of the degrees of the two polynomials. The coefficients of the new polynomial are simply the addition or subtraction of the coefficients of same order.

Multiplying two polynomials yields a polynomial whose degree is the product of the degrees of the two polynomials. If $\{a_0, \ldots, a_n\}$ and $\{b_0, \ldots, b_n\}$ are the coefficients of two polynomials, the coefficients of the product polynomial are given by:

$$c_k = \sum_{i+j=k} a_i b_j \quad \text{for } k = 0, \ldots, n+m. \tag{2.13}$$

In equation 2.13, the coefficients $a_k$ are treated as zero if $k > n$. Similarly, the coefficients $n_k$ are treated as zero if $k > m$.

Dividing a polynomial by another is akin to integer division with remainder. In other words, equation 2.14,

$$P(x) = Q(x) \cdot T(x) + R(x), \tag{2.14}$$

uniquely defines the two polynomials $Q(x)$, the quotient, and $R(x)$, the remainder, for any two given polynomials $P(x)$ and $T(x)$. The algorithm is similar to the algorithm taught in elementary school for dividing integers [Knudth 2].

### 2.2.2 Polynomials—General Implementation

As we have seen, a polynomial is uniquely defined by its coefficients. Thus, the creation of a new polynomial instance must have the coefficients given. Our implementation assumes that the first element of the array containing the coefficients is the coefficient of the constant term; the second element, the coefficient of the linear term ($x$); and so on.

The method `value` evaluates the polynomial at the supplied argument. This method implements equation 2.7.

The methods `derivative` and `integral` each return a new instance of a polynomial. The method `integral:` must have an argument specifying the value of the integral of the polynomial at zero. A convenience method `integral` without argument is equivalent to calling the method `integral` with argument 0.

The implementation of polynomial arithmetic is rarely used in numerical computation, though. It is, however, a nice example to illustrate a technique called *double dispatching* (double dispatching is described in the appendix, see Section ???). The need for double dispatching comes from allowing an operation between objects of different natures. In the case of polynomials, operations can be defined between two polynomials or between a number and a polynomial. In short, double dispatching allows one to identify the correct method based on the type of the two arguments.

### 2.2.3 Polynomials—Smalltalk Implementation

Being a special case of a function, a polynomial must implement the behavior of functions as discussed in Section 2.1.1. Code Example 2.2 describes how to use the class `DhbPolynomial`.

**Code Example 2.2**

```
| polynomial |
polynomial := DhbPolynomial coefficients: #(2 -3 1).
polynomial value: 1.
```

This code creates an instance of the class `DhbPolynomial` by giving the coefficient of the polynomial. In this example, the polynomial is $x^2 - 3x + 2$. The final line of the code computes the value of the polynomial at $x = 1$.

Code Example 2.3 shows how to manipulate polynomials in symbolic form.

**Code Example 2.3**

```
| pol1 pol2 polynomial polD polI|
pol1:= DhbPolynomial
coefficients: #(2 -3 1).
pol2:= DhbPolynomial coefficients: #(-3 7 2 1).
polynomial = pol1 * pol2.
polD := polynomial derivative.
polI := polynomial integral.
```

The first line creates the polynomial given in equation 2.6. The second line creates the polynomial $x^3 + 2x^2 + 7x - 3$. The third line of the code creates a new polynomial, the product of the first two. The last two lines create two polynomials: the derivative and the integral, respectively, of the polynomial created in the third line.

Listing 2.3 shows the Smalltalk implementation of the class `DhbPolynomial`.

A beginner may have been tempted to make `DhbPolynomial` a subclass of `Array` to spare the need for an instance variable. This is of course quite wrong. An array is a subclass of `Collection`. Most methods implemented or inherited by `Array` have nothing to do with the behavior of a polynomial as a mathematical entity.

Thus, a good choice is to make the class `DhbPolynomial` a subclass of Object. It has a single-instance variable, an instance of class `Array` containing the coefficients of the polynomial.

It is always a good idea to implement a method `printOn:` for each class. This method is used by many system utilities to display an object in readable form, particularly the debugger and the inspectors. The standard method defined for all objects simply displays the name of the class. Thus, it is hard to decide whether two different variables are pointing to the same object. Implementing a method `printOn:` for each class created is a good general practice, that can make life as a

Smalltalker much easier. It allows displaying parameters particular to each instance so that the instances can be easily identified. It may also be used in quick print on the `Transcript`.

Working with indices in Smalltalk is somewhat awkward for mathematical formulas because resulting the code is quite verbose. In addition a mathematician using Smalltalk for the first time may be disconcerted with all indices starting at 1 instead of 0. Smalltalk, however, has very powerful iteration methods, which largely compensate for the odd index choice—odd for a mathematician, that is. In fact, an experienced Smalltalker seldom uses indices explicitly as Smalltalk provides powerful iterator methods.

The method `value:` uses the Smalltalk iteration method `inject:into:` (see Section ???). Using this method requires storing the coefficients in reverse order because the first element fed into the method `inject:into:` corresponds to the coefficient of the largest power of $x$. It would certainly be quite inefficient to reverse the order of the coefficients at each evaluation. Since this requirement also simplifies the computation of the coefficients of the derivative and of the integral, reversing the coefficients is done in the creation method to make things transparent.

The methods `derivative` and `integral` return a new instance of the class `DhbPolynomial`. They do not modify the object receiving the message. This is also true for all operations between polynomials. The methods `derivative` and `integral` use the method `collect:`, returning a collection of the values returned by the supplied block closure at each argument (see Section ???).

The method `at:` allows one to retrieve a given coefficient. To ease readability of the multiplication and division methods, the method `at:` has been defined to allow for indices starting at zero. In addition, this method returns zero for any index larger than the polynomial's degree, which allows being lax with the index range. In particular, equation 2.13 can be coded exactly as it is.

The arithmetic operations between polynomials are implemented using double dispatching. This is a general technique widely used in Smalltalk (and all other languages with dynamic typing) consisting of selecting the proper method based on the type of the supplied arguments. Double dispatching is explained in section ???.

Note: Because Smalltalk is a dynamically typed language, my implementation of polynomial is also valid for polynomials with complex coefficients. This is not the case in Java, which requires explicit typing.

**Listing 2.3 Smalltalk implementation of the polynomial class**

| | |
|---|---|
| *Class* | `DhbPolynomial` |
| *Subclass of* | `Object` |
| *Instance variable names:* | `coefficients` |

*Class Methods*

**coefficients:**

```
anArray

^self new initialize: anArray reverse
```

*Instance Methods*

***** aNumberOrPolynomial

```
^aNumberOrPolynomial timesPolynomial: self
```

**+** aNumberOrPolynomial

```
^aNumberOrPolynomial addPolynomial: self
```

**-** aNumberOrPolynomial

```
^aNumberOrPolynomial subtractToPolynomial: self
```

**/** aNumberOrPolynomial

```
^aNumberOrPolynomial dividingPolynomial: self
```

**addNumber:** aNumber

```
| newCoefficients |
newCoefficients := coefficients reverse.
newCoefficients at: 1 put: newCoefficients first + aNumber.
^self class new: newCoefficients
```

**addPolynomial:** aPolynomial

```
^self class new: ( ( 0 to: (self degree max: aPolynomial degree))
            collect: [ :n | ( aPolynomial at: n) + ( self at: n)])
```

**at:** anInteger

```
^anInteger < coefficients size
    ifTrue: [ coefficients at: ( coefficients size - anInteger)]
    ifFalse:[ 0]
```

**coefficients**

```
^coefficients deepCopy
```

**degree**

```
^coefficients size - 1
```

**derivative**

```
| n |
n := coefficients size.
^self class new: ( ( coefficients collect: [ :each | n := n - 1.
            each * n]) reverse copyFrom: 2 to: coefficients size)
```

**dividingPolynomial:** aPolynomial

```
^( self dividingPolynomialWithRemainder: aPolynomial) first
```

**dividingPolynomialWithRemainder:** aPolynomial

```
| remainderCoefficients quotientCoefficients n m norm
                                                    quotientDegree |
n := self degree.
m := aPolynomial degree.
quotientDegree := m - n.
quotientDegree < 0
    ifTrue: [ ^Array with: ( self class new: #(0)) with:
                                                      aPolynomial].
quotientCoefficients := Array new: quotientDegree + 1.
remainderCoefficients := ( 0 to: m) collect: [ :k | aPolynomial
                                                          at: k].
norm := 1 / coefficients first.
quotientDegree to: 0 by: -1
    do: [ :k | | x |
          x := ( remainderCoefficients at: n + k + 1) * norm.
          quotientCoefficients at: (quotientDegree + 1 - k) put:
                                                               x.
          (n + k - 1) to: k by: -1
            do: [ :j |
            remainderCoefficients at: j + 1 put:
                        ( ( remainderCoefficients at: j + 1) -
                                            (x * (self at: j - k)))
                ].
        ].
^Array with: ( self class new: quotientCoefficients reverse)
       with: ( self class new: ( remainderCoefficients copyFrom:
                                                        1 to: n))
```

**initialize:** anArray

```
coefficients := anArray.
^self
```

**integral**

```
^self integral: 0
```

**integral:** aValue

```
| n |
n := coefficients size + 1.
^self class new: ( ( coefficients collect: [ :each | n := n - 1.
                              each / n]) copyWith: aValue) reverse
```

**printOn:** aStream

```
| n firstNonZeroCoefficientPrinted  |
n := 0.
firstNonZeroCoefficientPrinted := false.
coefficients reverse do:
    [ :each |
      each = 0
        ifFalse:[ firstNonZeroCoefficientPrinted
                        ifTrue: [ aStream space.
                                     each < 0
                                        ifFalse:[ aStream
                                                     nextPut: $+].
                                     aStream space.
                                 ]
                        ifFalse:[ firstNonZeroCoefficientPrinted
                                                          := true].
                      ( each = 1 and: [ n > 0])
                        ifFalse:[ each printOn: aStream].
                      n > 0
                        ifTrue: [ aStream nextPutAll: ' X'.
                                     n > 1
                                        ifTrue: [ aStream
                                                      nextPut: $^.
                                                  n printOn:
                                                       aStream.
                                                ].
                                ].
                ].
     n := n + 1.
    ].
```

**subtractToPolynomial:** aPolynomial

```
^self class new: ( ( 0 to: (self degree max: aPolynomial degree))
            collect: [ :n | ( aPolynomial at: n) - ( self at: n)])
```

**timesNumber:** aNumber

```
^self class new: ( coefficients reverse collect: [ :each | each *
                                                          aNumber])
```

**timesPolynomial:** aPolynomial

```
| productCoefficients degree|
degree := aPolynomial degree + self degree.
productCoefficients := ( degree to: 0 by: -1)
        collect:[ :n | | sum |
                  sum := 0.
                  0 to: (degree - n)
                    do: [ :k | sum := (self at: k) * (aPolynomial
                                    at: ( degree - n - k)) + sum].
                  sum
                ].
^self class new: productCoefficients
```

**value:** aNumber

```
^coefficients inject: 0 into: [ :sum :each | sum * aNumber +
                                                        each]
```

Listing 2.4 shows the listing of the methods used by the class Number as part of the double dispatching of the arithmetic operations on polynomials.

**Listing 2.4 Method of class Number related to polynomials**

| | |
|---|---|
| *Class* | Number |
| *Subclass of* | Magnitude |

***Instance Methods***

**addPolynomial:** aPolynomial

```
^aPolynomial addNumber: self
```

**dividingPolynomial:** aPolynomial

```
^aPolynomial timesNumber: (1 / self)
```

**subtractToPolynomial:** aPolynomial

```
^aPolynomial addNumber: self negated
```

**timesPolynomial:** aPolynomial

```
^aPolynomial timesNumber: self
```

### 2.2.4 Polynomials—Java implementation

Listing 2.5 shows the Java implementation of the class `PolynomialFunction`. Code Example 2.4 describes how to use the class `PolynomialFunction`.

**Code Example 2.4**

```
double[] coefficients = {2, -3, 1};
PolynomialFunction poly = new PolynomialFunction( coefficients);
double p1 = polynomial.value( 1).
```

First the coefficients of the polynomial $x^2 - 3x + 2$ are defined in a variable. The next line creates a new instance of the class `PolynomialFunction` with these coefficients. The final line of the code calculates the value of the polynomial at $x = 1$.

For the same reason explained in the Smalltalk section, the class `PolynomialFunction` is a subclass of `Object`. It implements the interface `OneVariableFunction` (see Section 2.1.2) indicating that this class must implement the method value. Thus, a polynomial can be used by several other classes defined hereafter in this book.

The Java implementation uses indices since array elements are referred to by indices starting at zero. This approach makes a direct comparison with the mathematical expression easier than in Smalltalk. Nevertheless, the method `coefficient` was created to access coefficients for any nonnegative index. If the index is larger than the polynomial's degree, this routine returns zero. Using this method greatly simplifies the code for adding and multiplying polynomials. The method `coefficient` corresponds to the method `at:` in the Smalltalk implementation.

Polynomial arithmetic is implemented with special methods since Java does not allow overloading the arithmetic operators. As Java methods and their arguments are typed, a unique method must be written for each argument type. Therefore, there is no need for double dispatching.[6]

Finally, the method `toString()` is implemented for the same reasons than a method `printOn:` was implemented in Smalltalk.

**Note:** Also the three methods `deflate` and `roots` (`roots` having two variants) are discussed in Section 5.4.2.

---

**Listing 2.5 Java implementation of the polynomial class**

```
package DhbFunctionEvaluation;

import java.util.Vector;
import java.util.Enumeration;
import DhbInterfaces.OneVariableFunction;
import DhbIterations.NewtonZeroFinder;
import DhbFunctionEvaluation.DhbMath;
```

6. On the other hand, the readability of code using polynomial arithmetic is much better in Smalltalk than in Java.

```
// Mathematical polynomial:
// c[0] + c[1] * x + c[2] * x^2 + ....

// @author Didier H. Besset

public class PolynomialFunction implements OneVariableFunction
{

    // Polynomial coefficients.

    private double[] coefficients;

// Constructor method.
// @param coeffs polynomial coefficients.

public PolynomialFunction( double[] coeffs)
{
    coefficients = coeffs;
}

// @param r double    number added to the polynomial.
// @return DhbFunctionEvaluation.PolynomialFunction

public PolynomialFunction add( double r)
{
    int n = coefficients.length;
    double coef[] = new double[n];
    coef[0] = coefficients[0] + r;
    for ( int i = 1; i < n; i++)
        coef[i] = coefficients[i];
    return new PolynomialFunction( coef);
}

// @param p DhbFunctionEvaluation.PolynomialFunction
// @return DhbFunctionEvaluation.PolynomialFunction

public PolynomialFunction add( PolynomialFunction p)
{
    int n = Math.max( p.degree(), degree()) + 1;
    double[] coef = new double[n];
    for ( int i = 0; i < n; i++ )
        coef[i] = coefficient(i) + p.coefficient(i);
    return new PolynomialFunction( coef);
}

// Returns the coefficient value at the desired position
// @param n int     the position of the coefficient to be returned
```

```
// @return double the coefficient value
// @version 1.2

public double coefficient( int n)
{
    return n < coefficients.length ? coefficients[n] : 0;
}

// @param r double    a root of the polynomial (no check made).
// @return PolynomialFunction the receiver divided by polynomial (x - r).

public PolynomialFunction deflate( double r)
{
    int n = degree();
    double remainder = coefficients[n];
    double[] coef = new double[n];
    for ( int k = n - 1; k >= 0; k--)
    {
        coef[k] = remainder;
        remainder = remainder * r + coefficients[k];
    }
    return new PolynomialFunction( coef);
}

// Returns degree of this polynomial function
// @return int degree of this polynomial function

public int degree()
{
    return coefficients.length - 1;
}

// Returns the derivative of the receiver.
// @return PolynomialFunction derivative of the receiver.

public PolynomialFunction derivative()
{
    int n = degree();
    if ( n == 0 )
    {
        double coef[] = {0};
        return new PolynomialFunction( coef);
    }
    double coef[] = new double[n];
    for ( int i = 1; i <= n; i++)
        coef[i-1] = coefficients[i]*i;
```

```
        return new PolynomialFunction( coef);
    }

    // @param r double
    // @return DhbFunctionEvaluation.PolynomialFunction

    public PolynomialFunction divide( double r)
    {
        return multiply( 1 / r);
    }

    // @param r double
    // @return DhbFunctionEvaluation.PolynomialFunction

    public PolynomialFunction divide( PolynomialFunction p)
    {
        return divideWithRemainder(p)[0];
    }


    // @param r double
    // @return DhbFunctionEvaluation.PolynomialFunction

    public PolynomialFunction[] divideWithRemainder( PolynomialFunction p)
    {
        PolynomialFunction[] answer = new PolynomialFunction[2];
        int m = degree();
        int n = p.degree();
        if ( m < n )
        {
            double[] q = {0};
            answer[0] = new PolynomialFunction( q);
            answer[1] = p;
            return answer;
        }
        double[] quotient = new double[ m - n + 1];
        double[] coef = new double[ m + 1];
        for ( int k = 0; k <= m; k++ )
            coef[k] = coefficients[k];
        double norm = 1 / p.coefficient( n);
        for ( int k = m - n; k >= 0; k--)
        {
            quotient[k] = coef[ n + k] * norm;
            for ( int j = n + k - 1; j >= k; j--)
                coef[j] -= quotient[k] * p.coefficient(j-k);
        }
```

```
        double[] remainder = new double[n];
        for ( int k = 0; k < n; k++)
            remainder[k] = coef[k];
        answer[0] = new PolynomialFunction( quotient);
        answer[1] = new PolynomialFunction( remainder);
        return answer;
    }

    // Returns the integral of the receiver having the value 0 for X = 0.
     * @return PolynomialFunction integral of the receiver.

    public PolynomialFunction integral( )
    {
        return integral( 0);
    }

    // Returns the integral of the receiver having the specified value for X = 0.
    // @param value double    value of the integral at x=0
    // @return PolynomialFunction integral of the receiver.

    public PolynomialFunction integral( double value)
    {
        int n = coefficients.length + 1;
        double coef[] = new double[n];
        coef[0] = value;
        for ( int i = 1; i < n; i++)
            coef[i] = coefficients[i-1]/i;
        return new PolynomialFunction( coef);
    }

    // @param r double
    // @return DhbFunctionEvaluation.PolynomialFunction

    public PolynomialFunction multiply( double r)
    {
        int n = coefficients.length;
        double coef[] = new double[n];
        for ( int i = 0; i < n; i++)
            coef[i] = coefficients[i] * r;
        return new PolynomialFunction( coef);
    }

    // @param p DhbFunctionEvaluation.PolynomialFunction
    // @return DhbFunctionEvaluation.PolynomialFunction

    public PolynomialFunction multiply( PolynomialFunction p)
```

```
{
    int n = p.degree() + degree();
    double[] coef = new double[n + 1];
    for ( int i = 0; i <= n; i++)
    {
        coef[i] = 0;
        for ( int k = 0; k <= i; k++)
            coef[i] += p.coefficient(k) * coefficient(i-k);
    }
    return new PolynomialFunction( coef);
}

// @return double[]

public double[] roots()
{
    return roots( DhbMath.defaultNumericalPrecision());
}

// @param desiredPrecision double
// @return double[]

public double[] roots( double desiredPrecision)
{
    PolynomialFunction dp = derivative();
    double start = 0;
    while ( Math.abs( dp.value( start)) < desiredPrecision )
        start = Math.random();
    PolynomialFunction p = this;
    NewtonZeroFinder rootFinder = new NewtonZeroFinder( this, dp, start);
    rootFinder.setDesiredPrecision( desiredPrecision);
    Vector rootCollection = new Vector( degree());
    while ( true)
    {
        rootFinder.evaluate();
        if ( !rootFinder.hasConverged() )
            break;
        double r = rootFinder.getResult();
        rootCollection.addElement( new Double( r));
        p = p.deflate( r);
        if ( p.degree() == 0 )
            break;
        rootFinder.setFunction( p);
        try { rootFinder.setDerivative( p.derivative());}
                catch ( IllegalArgumentException e) {};
    }
```

```
        double[] roots = new double[ rootCollection.size()];
        Enumeration e = rootCollection.elements();
        int n = 0;
        while ( e.hasMoreElements() )
        {
            roots[n++] = ( (Double) e.nextElement()).doubleValue();
        }
        return roots;
    }

    // @param p DhbFunctionEvaluation.PolynomialFunction
    // @return DhbFunctionEvaluation.PolynomialFunction

    public PolynomialFunction subtract( double r)
    {
        return add( -r);
    }

    // @return DhbFunctionEvaluation.PolynomialFunction
    // @param p DhbFunctionEvaluation.PolynomialFunction

    public PolynomialFunction subtract( PolynomialFunction p)
    {
        int n = Math.max( p.degree(), degree()) + 1;
        double[] coef = new double[n];
        for ( int i = 0; i < n; i++ )
            coef[i] = coefficient(i) - p.coefficient(i);
        return new PolynomialFunction( coef);
    }

    // eturns a string representing the receiver

    public String toString()
    {
        StringBuffer sb = new StringBuffer();
        boolean firstNonZeroCoefficientPrinted = false;
        for ( int n = 0; n < coefficients.length; n++)
        {
            if ( coefficients[n] != 0 )
            {
                if ( firstNonZeroCoefficientPrinted)
                    sb.append( coefficients[n] > 0 ? " + " : " ");
                else
                    firstNonZeroCoefficientPrinted = true;
```

```
            if ( n == 0 || coefficients[n] != 1)
                  sb.append( Double.toString( coefficients[n]) );
              if ( n > 0 )
                  sb.append( " X^"+n);
          }
      }
      return sb.toString();
  }

  // Returns the value of the polynomial for the specified variable value.
  // @param x double    value at which the polynomial is evaluated
  // @return double polynomial value.

  public double value( double x)
  {
      int n = coefficients.length;
      double answer = coefficients[--n];
      while ( n > 0 )
          answer = answer * x + coefficients[--n];
      return answer;
  }

  // Returns the value and the derivative of the polynomial
  // for the specified variable value in an array of two elements
  // @version 1.2
  // @param x double    value at which the polynomial is evaluated
  // @return double[0]   the value of the polynomial
  // @return double[1]   the derivative of the polynomial

  public double[] valueAndDerivative( double x)
  {
      int n = coefficients.length;
      double[] answer = new double[2];
      answer[0] = coefficients[--n];
      answer[1] = 0;
      while ( n > 0 )
      {
          answer[1] = answer[1] * x + answer[0];
          answer[0] = answer[0] * x + coefficients[--n];
      }
      return answer;
  }
  }
```

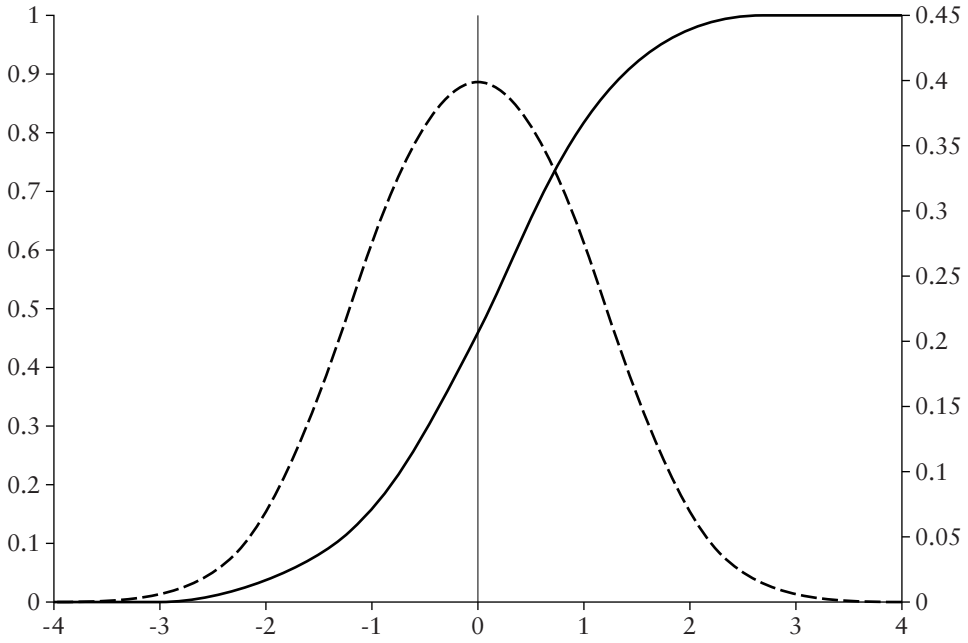


**FIG. 2.3** The error function (solid line) and the normal distribution (dotted line)

## 2.3 Error Function

The error function is the integral of the normal distribution. The error function is used in statistics to evaluate the probability of finding a measurement larger than a given value when the measurements are distributed according to a normal distribution. Figure 2.3 shows the familiar bell-shaped curve of the probability density function of the normal distribution (dotted line) together with the error function (solid line).

In medical sciences, one calls *centile* the value of the error function expressed as a percentage. For example, obstetricians consider whether the weight at birth of the first-born child is located below the 10th centile or above the 90th centile to assess a risk factor for a second pregnancy.[7]

### 2.3.1 Mathematical Definitions

Because it is the intergral of the normal distribution, the error function, $\mathrm{erf}(x)$, gives the probability of finding a value lower than $x$ when the values are distributed according to a normal distribution with a mean of zero and a standard deviation

7. see footnote 9 in Chapter 1.

of 1. The mean and standard deviation are explained in Section ???. This probability is expressed by the following integral[8]:

$$\text{erf}(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{t^2}{2}} dt. \tag{2.15}$$

The result of the error function lies between 0 and 1.

One could carry out the integral numerically, but several good approximations exist. Equation 2.16 is taken from [Abramovitz & Stegun].

$$\text{erf}(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} \sum_{i=1}^{5} a_i r(x)^i \quad \text{for } x \geq 0, \tag{2.16}$$

where

$$r(x) = \frac{1}{1 - 0.2316419x} \tag{2.17}$$

and

$$\begin{cases} a_1 = & 0.31938153 \\ a_2 = & -0.356563782 \\ a_3 = & 1.7814779372 \\ a_4 = & -1.821255978 \\ a_5 = & 1.330274429 \end{cases}. \tag{2.18}$$

The error on this formula is better than $7.5 \times 10^{-8}$ for positive $x$. To compute the value for negative values, one uses the fact that

$$\text{erf}(x) = 1 - \text{erf}(-x). \tag{2.19}$$

When dealing with a general Gaussian distribution with average $\mu$ and standard deviation $\sigma$ it is convenient to define a generalized error function as

$$\text{erf}(x; \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} \int_{-\infty}^{x} e^{-\frac{(x-\mu)^2}{2\sigma^2}} dt. \tag{2.20}$$

A simple change of variable in the integral shows that the generalized error function can be obtained from the error function as

$$\text{erf}(x; \mu, \sigma) = \text{erf}(\frac{x - \mu}{\sigma}). \tag{2.21}$$

Thus, one can compute the probability of finding a measurement $x$ within the interval $[\mu - t \cdot \sigma, \mu + t \cdot \sigma]$ when the measurements are distributed according to

8. In [Abramovitz & Stegun] and [Press *et al.*], the error function is defined as

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_{0}^{x} e^{-\frac{t^2}{2}} dt.$$

a Gaussian distribution with average $\mu$ and standard deviation $\sigma$:

$$\text{Prob}\left(\frac{|x-\mu|}{\sigma} \leq t\right) = 2 \cdot \text{erf}(t) - 1. \quad \text{for } t \geq 0. \tag{2.22}$$

Now we can answer the problem of deciding whether a pregnant woman needs special attention during her second pregnancy. Let the weight at birth of her first child be 2.85 kg, and let the duration of her first pregnancy be 39 weeks. In this case, measurements over a representative sample of all births yielding healthy babies have an average of 3.39 kg and a standard deviation of 0.44 kg.[9]. The probability of a second child having a weight at birth less than that of the woman's first child is

$$\begin{aligned}\text{Prob}(\text{weight} \leq 2.85 \text{ kg}) &= \text{erf}\left(\frac{2.85 - 3.39}{0.44}\right), \\ &= 11.2\%.\end{aligned}$$

According to current practice, this second pregnancy does not require special attention.

### 2.3.2 Error Function—Smalltalk Implementation

The error function is implemented as a single method for the class `Number`. Code Example 2.5 shows how one can easily compute the centile of our preceding example.

**Code Example 2.5**

```
| weight average stDev centile |
weight := 2.85.
average := 3.39.
stDev := 0.44.
centile := ( ( weight - average) / stDev) erf * 100.
```

If you want to compute the probability for a measurement to lie within 3 standard deviations from its mean, you need to evaluate the expression given in Code Example 2.6 using equation 2.22.

**Code Example 2.6**

```
3 errorFunction *2 - 1
```

If you need to use the error function as a function, you must use it inside a block closure. In this case you would define a function object as indicated in Code Example 2.7.

9. This is the practice at the department of obstetrics and gynecology of the Chelsea and Westminster Hospital of London. The numbers are reproduced with permission of Prof. P. J. Steer.

**Code Example 2.7**

```
| errorFunction |
errorFunction := [ :x | x errorFunction].
```

Listing 2.6 shows the Smalltalk implementation of the error function.

Smalltalk allows us to extend existing classes. Thus, the public method to evaluate the error function is implemented as a method of the base class `Number`. This method uses the class `DhbErfApproximation` to store the constants of equation 2.18 and evaluate the formula of equations 2.16 and 2.17. In our case, there is no need to create a separate instance of the class `DhbErfApproximation` at each time since all instances would actually be exactly identical. Thus, the class `DhbErfApproximation` is a singleton class. A *singleton class* is a class, that can only create a single instance [Gamma *et al.*]. Once the first instance is created, it is kept in a class instance variable. Any subsequent attempt to create an additional instance will return a pointer to the class instance variable holding the first created instance.

One could have implemented all of these methods as class methods to avoid the singleton class. In Smalltalk, however, one tends to reserve class method for behavior needed by the structural definition of the class. So, the use of a singleton class is preferable. A more detailed discussion of this topic can be found in [Alpert *et al.*].

---

**Listing 2.6 Smalltalk implementation of the Error function**

| | |
|---|---|
| *Class* | `Number` |
| *Subclass of* | `Magnitude` |

***Instance Methods***

**errorFunction**

```
^DhbErfApproximation new value: self
```

| | |
|---|---|
| *Class* | `DhbErfApproximation` |
| *Subclass of* | `Object` |
| *Instance variable names:* | `constant series norm` |
| *Class variable names:* | `UniqueInstance` |

***Class Methods***

**new**

```
UniqueInstance isNil
    ifTrue: [ UniqueInstance := super new.
              UniqueInstance initialize.
            ].
```

```
^UniqueInstance
```

*Instance Methods*

**initialize**

```
constant := 0.2316419.
norm := 1 / ( Float pi * 2) sqrt.
series := DhbPolynomial coefficients: #( 0.31938153 -0.356563782
                            1.781477937 -1.821255978 1.330274429).
```

**normal:** aNumber

```
^[ ( aNumber squared * -0.5) exp * norm]
        when: ExAll do: [ :signal | signal exitWith: 0]
```

**value:** aNumber

```
| t |
aNumber = 0
    ifTrue: [ ^0.5].
aNumber > 0
    ifTrue: [ ^1- ( self value: aNumber negated)].
aNumber < -20
    ifTrue: [ ^0].
t := 1 / (1 - (constant * aNumber)).
^( series value: t) * t * (self normal: aNumber)
```

---

## 2.3.3 Error Function—Java Implementation

Unfortunately, Java does not allow for extension of existing classes. Thus, a new class must be created. All methods are implemented as static methods since there is no instance dependency. Static methods in Java have little in common with Smalltalk class methods. In particular, there is no inheritance of static methods. Static methods indicate that the code is not instance-dependent and are used to generate compiler and run time optimization. Thus, a singleton class is not needed in this case.

Since the error function is almost always used in conjunction with statistical analysis, we have implemented the method as a static method of the class `NormalDistribution` discussed in section ???. Thus, computing the error function of 2 must be coded as shown in Code Example 2.8.

**Code Example 2.8**

```
double x = NormalDistribution.errorFunction(2)
```

If you want to compute the probability for a measurement to lie within 3 standard deviations from its mean, you need to evaluate the expression given in Code Example 2.9.

**Code Example 2.9**

```
double y = 2 * NormalDistribution.errorFunction(3)- 1
```

Listing 2.7 shows the Java implementation of the error function.

The class `NormalDistribution` is tagged as `final` because it does not make sense that this class has any subclass.

---

**Listing 2.7 Java implementation of the Error function (partial listing)**

```
package DhbStatistics;

import DhbFunctionEvaluation.PolynomialFunction;

// Normal distribution, a.k.a. Gaussian distribution.

public final class NormalDistribution extends ProbabilityDensityFunction
{
  private static double baseNorm = Math.sqrt( 2 * Math.PI);

  // Series to compute the error function.

  private static PolynomialFunction errorFunctionSeries;

  // Constant needed to compute the argument to the error function series.

  private static double errorFunctionConstant = 0.2316419;

// @return error function for the argument.
// @param x double

public static double errorFunction ( double x)
{
  if ( errorFunctionSeries == null )
  {
    double[] coeffs = { 0.31938153, -0.356563782, 1.781477937,
    -1.821255978, 1.330274429};
    errorFunctionSeries = new PolynomialFunction( coeffs);
  }
  if ( x > 0 )
    return 1 - errorFunction( -x);
  double t = 1 / (1 - errorFunctionConstant * x);
  return t * errorFunctionSeries.value( t) * normal( x);
}

// @return the density probability function for a (0,1) normal distribution.
```

```
// @param x double  value for which the probability is evaluated.

static public double normal( double x)
{
  return Math.exp( -0.5 * x * x) / baseNorm;
}
}
```

## 2.4 Gamma Function

The gamma function is used in many mathematical functions. In this book, it is needed to compute the normalization factor of several probability density functions (see Sections ??? and ???) and to compute the beta function (see Section 2.5).

### 2.4.1 Mathematical Definitions

The gamma function is defined by the following integral, called Euler's integral[10]:

$$\Gamma(x) = \int_0^\infty t^x e^{-t} dt. \tag{2.23}$$

From equation 2.23, a recurrence formula can be derived:

$$\Gamma(x+1) = x \cdot \Gamma(x) \tag{2.24}$$

The value of the gamma function can be computed for special values of $x$:

$$\begin{cases} \Gamma(1) = 1 \\ \Gamma(2) = 1 \end{cases}. \tag{2.25}$$

From equations 2.24 and 2.25, the well-known relation between the value of the Gamma function for positive integers and the factorial can be derived

$$\Gamma(n) = (n-1)! \quad \text{for } n > 0. \tag{2.26}$$

The most precise approximation for the Gamma function is given by a formula discovered by Lanczos [Press *et al.*]:

$$\Gamma(x) \approx e^{\left(x+\frac{5}{2}\right)} \left(x + \frac{5}{2}\right) \frac{\sqrt{2\pi}}{x} \left( c_0 + \sum_{n=1}^{6} \frac{c_n}{x+n} + \epsilon \right), \tag{2.27}$$

where

10. Leonard Euler, to be precise, as the Euler family produced many mathematicians.

$$\begin{cases} c_0 = & 1.000000000190015 \\ c_1 = & 76.18009172947146 \\ c_2 = & \text{-}86.50532032941677 \\ c_3 = & 24.01409824083091 \\ c_4 = & \text{-}1.231739572450155 \\ c_5 = & 1.208650973866179 \cdot 10^{-3} \\ c_6 = & \text{-}5.395239384953 \cdot 10^{-6} \end{cases} . \tag{2.28}$$

This formula approximates $\Gamma(x)$ for $x > 1$ with $\epsilon < 2 \cdot 10^{-10}$. Actually, this remarkable formula can be used to compute the gamma function of any complex number $z$ with $\Re(z) > 1$ to the quoted precision. Combining Lanczos's formula with the recurrence formula 2.24 is sufficient to compute values of the gamma function for all positive numbers.

For example, $\Gamma\left(\frac{3}{2}\right) = \frac{\sqrt{\pi}}{2} = 0.886226925452758$, whereas Lanczos's formula yields the value 0.886226925452754—that is, an absolute error of $4 \cdot 10^{-15}$. The corresponding relative precision is almost equal to the floating-point precision of the machine on which this computation was made.

Although this value is seldom used, the value of the gamma function for negative noninteger numbers can be computed using the reflection formula

$$\Gamma(x) = \frac{\pi}{\Gamma(1-x)\sin \pi x}. \tag{2.29}$$

In summary, the algorithm to compute the gamma function for any argument goes as follows:

1. If $x$ is a nonpositive integer ($x \leq 0$), raise an exception.
2. If $x$ is smaller than or equal to 1 ($x \leq 1$), use the recurrence formula (equation 2.24).
3. If $x$ is negative ($x < 0$, but noninteger), use the reflection formula (equation 2.29).
4. Otherwise, use Lanczos's formula (equation 2.27).

One can see from the leading term of Lanczos's formula that the gamma function raises faster than an exponential. Thus, evaluating the gamma function for numbers larger than a few hundred will exceed the capacity of the floating-number representation on most machines. For example, the maximum exponent of a double-precision IEEE floating-point number is 1024. Evaluating directly the expression:

$$\frac{\Gamma(460.5)}{\Gamma(456.3)} \tag{2.30}$$

will fail since $\Gamma(460.5)$ is larger than $10^{1024}$. Thus, its evaluation yields a floating-point overflow exception. It is therefore recommended to use the logarithm of the gamma function whenever it is used in quotients involving large numbers. The

expression of equation 2.30 is then evaluated as:

$$\exp\left[\ln\Gamma(460.5) - \ln\Gamma(456.3)\right], \tag{2.31}$$

which yields the result $1.497 \cdot 10^{11}$. That result fits comfortably within the floating-point representation.

For similar reasons, the leading factors of Lanczos's formula are evaluated using logarithms in both implementations.

### 2.4.2 Gamma Function—Smalltalk Implementation

Like the error function, the gamma function is implemented as a single method of the class `Number`. Thus, computing the gamma function of 2.5 is simply coded as shown in Code Example 2.10.

**Code Example 2.10**

```
2.5 gamma
```

To obtain the logarithm of the gamma function, you need to evaluate the expression given in Code Example 2.11.

**Code Example 2.11**

```
2.5 logGamma
```

Listing 2.8 shows the Smalltalk implementation of the gamma function.

Here, the gamma function is implemented with two methods: one for the class `Integer` and one for the class `Float`. Otherwise, the scheme to define the gamma function is similar to that of the error function. Please refer to Section 2.3.2 for detailed explanations.

Since the method `factorial` is already defined for integers in the base classes, the gamma function has been defined using equation 2.26 for integers. An error is generated if one attempts to compute the gamma function for nonpositive integers. The class `Number` delegates the computation of Lanczos's formula to a singleton class. This is used by the noninteger subclasses of `Number`: `Float` and `Fraction`.

The execution time to compute the gamma function for a floating argument is given in Table 1.1 in Section 1.6.

---

**Listing 2.8 Smalltalk implementation of the gamma function**

| | |
|---|---|
| *Class* | `Integer` |
| *Subclass of* | `Number` |

### *Instance Methods*

**gamma**

```
self > 0
    ifFalse:[ ^self error: 'Attempt to compute the Gamma function
                                        of a non-positive integer'].
^( self - 1) factorial
```

*Class* Number

*Subclass of* Magnitude

### *Instance Methods*

**gamma**

```
^self > 1
    ifTrue: [ ^DhbLanczosFormula new gamma: self]
    ifFalse:[ self < 0
                    ifTrue: [ Float pi / ( ( Float pi * self) sin
                                            * ( 1 - self) gamma)]
                    ifFalse:[ ( DhbLanczosFormula new gamma:
                                              (self + 1)) / self]
            ]
```

**logGamma**

```
^self > 1
    ifTrue: [ DhbLanczosFormula new logGamma: self]
    ifFalse:[ self > 0
                    ifTrue: [ ( DhbLanczosFormula new logGamma:
                                          (self + 1) ) - self ln ]
                    ifFalse: [ ^self error: 'Argument for the log
                                 gamma function must be positive']
            ]
```

*Class* DhbLanczosFormula

*Subclass of* Object

*Instance variable names:* coefficients sqrt2Pi

*Class variable names:* UniqueInstance

### *Class Methods*

**new**

```
UniqueInstance isNil
    ifTrue: [ UniqueInstance := super new.
```

```
                    UniqueInstance initialize.
            ].
^UniqueInstance
```

*Instance Methods*

**gamma:** aNumber

```
^( self leadingFactor: aNumber) exp * ( self series: aNumber) *
                                                    sqrt2Pi / aNumber
```

**initialize**

```
sqrt2Pi := ( Float pi * 2) sqrt.
coefficients := #( 76.18009172947146 -86.50532032941677
       24.01409824083091 -1.231739572450155 0.1208650973866179e-2
       -0.5395239384953e-5).
^self
```

**leadingFactor:** aNumber

```
| temp |
temp := aNumber + 5.5.
^( temp ln * ( aNumber + 0.5) - temp)
```

**logGamma:** aNumber

```
^( self leadingFactor: aNumber) + ( ( self series: aNumber) *
                                              sqrt2Pi / aNumber) ln
```

**series:** aNumber

```
| term |
term := aNumber.
^coefficients inject: 1.000000000190015
                    into: [ :sum :each | term := term + 1.
                                                 each / term + sum]
```

---

## 2.4.3 Gamma Function—Java Implementation

Like the error function, the gamma function is implemented as a static method in a special class. Thus, computing the gamma function of 2.5 is simply coded as shown in Code Example 2.12.

**Code Example 2.12**

```
double g = GammaFunction.gamma( 2.5)
```

To obtain the logarithm of the gamma function, you need to evaluate the expression given in Code Example 2.13.

**Code Example 2.13**

```
double lg = GammaFunction.logGamma( 2.5)
```

Listing 2.9 shows the Java implementation of the gamma function. (It also contains the method beta needed to compute the beta function discussed in Section 2.5.)

For the Java implementation, a specific class containing only static methods was created. This class also implements the beta function discussed in Section 2.5.

To prevent the unnecessary computation of a logarithm, the code for the gamma function and the logarithm of the gamma function have duplicated code.

Instead of raising an exception when the argument is a nonpositive integer, the function simply returns the system-defined value `Double.NaN`, a unique feature of Java to represent a number impossible to compute. This feature is quite handy because `Double.NaN` propagates itself in any subsequent expression without raising an exception. The object recuperating the end result can check for such value without the need to catch an exception.

Strangely enough, Java does not implement an Integer.NaN. Thus, the `method` factorial has no other choice than raising an exception when it is called with a negative integer argument.

---

**Listing 2.9 Java implementation of the gamma function**

```
package DhbFunctionEvaluation;

// Gamma function (Euler's integral).

// @author Didier H. Besset

public final class GammaFunction
{
    static double sqrt2Pi = Math.sqrt( 2 * Math.PI);
    static double[] coefficients = { 76.18009172947146,
                                    -86.50532032941677,
                                     24.01409824083091,
                                     -1.231739572450155,
                                      0.1208650973866179e-2,
                                     -0.5395239384953e-5};

// @return double       beta function of the arguments
// @param x double
// @param y double

public static double beta ( double x, double y)
```

```
{
    return Math.exp( logGamma( x) + logGamma( y) - logGamma( x + y));
}

// @return long    factorial of n
// @param n long

public static long factorial ( long n)
{
    return n < 2 ? 1 : n * factorial( n - 1);
}

// @return double      gamma function
// @param x double

public static double gamma ( double x)
{
    return x > 1
                ? Math.exp( leadingFactor(x)) * series(x) * sqrt2Pi / x
                : ( x > 0 ? gamma(x + 1) / x
                                : Double.NaN);
}

// @return double
// @param x double

private static double leadingFactor ( double x)
{
    double temp = x + 5.5;
    return Math.log( temp) * ( x + 0.5) - temp;
}

// @return double    logarithm of the beta function of the arguments
// @param x double
// @param y double

public static double logBeta ( double x, double y)
{
    return logGamma( x) + logGamma( y) - logGamma( x + y);
}

// @return double      log of the gamma function
// @param x double

public static double logGamma ( double x)
{
```

```
        return x > 1
                    ? leadingFactor(x) + Math.log( series(x) * sqrt2Pi / x)
                    : ( x > 0 ? logGamma(x + 1) - Math.log( x)
                                    : Double.NaN);
    }

    // @return double       value of the series in Lanczos formula.
    // @param x double

    private static double series( double x)
    {
        double answer = 1.000000000190015;
        double term = x;
        for ( int i = 0; i < 6; i++)
        {
            term += 1;
            answer += coefficients[i] / term;
        }
        return answer;
    }
    }
```

## 2.5 Beta Function

The beta function is directly related to the gamma function. In this book, the beta function is needed to compute the normalization factor of several probability density functions (see Sections ???, ???, and ???).

### 2.5.1 Mathematical Definitions

The beta function is defined by the integral:

$$B(x, y) = \int_0^1 t^{x-1}(1-t)^{y-1}dt. \tag{2.32}$$

The beta function is related to the gamma function with the following relation:

$$B(x, y) = \frac{\Gamma(x)\Gamma(y)}{\Gamma(x+y)}. \tag{2.33}$$

Thus, computation of the beta function is directly obtained from the gamma function. Because evaluating the gamma function might overflow the floating-point exponent (see the discussion at the end of Section 2.4.1), it is best to evaluate the prior formula using the logarithm of the gamma function.

### 2.5.2 Beta Function—Smalltalk Implementation

Like the error and gamma functions, the beta function is implemented as a single method of the class `Number`. Thus, computing the beta function of 2.5 and 5.5 is simply coded as shown in Code Example 2.14.

**Code Example 2.14**

```
2.5 beta: 5.5
```

Computing the logarithm of the beta function of 2.5 and 5.5 is simply coded as given in Code Example 2.15.

**Code Example 2.15**

```
2.5 logBeta: 5.5
```

Listing 2.10 shows the implementation of the beta function in Smalltalk.

---

**Listing 2.10 Smalltalk implementation of the beta function**

| | |
|---|---|
| *Class* | `Number` |
| *Subclass of* | `Magnitude` |

***Instance Methods***

**beta:** `aNumber`

```
^( self logBeta: aNumber) exp
```

**logBeta:** `aNumber`

```
^self logGamma + aNumber logGamma - ( self + aNumber) logGamma
```

---

### 2.5.3 Beta Function—Java Implementation

The Java implementation of the beta function consists of the two methods beta and `logBeta` in the class `GammaFunction` shown in Listing 2.9.

CHAPTER 3

# Interpolation

*On ne peut prévoir les choses qu'aprcs qu'elles sont arrivées.*[1]

—Eugcne Ionesco

Interpolation is a technique allowing the estimation of a function over the range covered by a set of points at which the function's values are known. These points are called the *sample* points. Interpolation is useful to compute a function whose evaluation is highly time-consuming: with interpolation it suffices to compute the function's values for a small number of well-chosen sample points. Then, evaluation of the function between the sample points can be made with interpolation.

Interpolation can also be used to compute the value of the inverse function—that is, to find a value $x$ such that $f(x) = c$, where $c$ is a given number, when the function is known for a few sample points bracketing the sought value. People often overlook this easy and direct computation of the inverse function.

Interpolation is often used interchangeably with extrapolation. This usage is not correct, however. Extrapolation is the task of estimating a function outside the range covered by the sample points. If no model exists for the data, extrapolation is just gambling. Methods exposed in this chapter should not be used for extrapolation.

Interpolation should not be mistaken with function (or curve) fitting. In the case of interpolation, the sample points purely determine the interpolated function. Function fitting allows constraining the fitted function independently from the sample points. As a result, fitted functions are more stable than interpolated functions, especially when the supplied values are subject to fluctuations coming from rounding or measurement errors. Fitting is discussed in Chapter ???.

1. One can predict things only after they have occurred.

## 3.1 General Remarks

There are several methods of interpolation. One difference is the type of function used. The other is the particular algorithm used to determine the function. For example, if the function is periodic, interpolation can be obtained by computing a sufficient number of coefficients of the Fourier series for that function.

In the absence of any information about the function, polynomial interpolation gives fair results. The function should not have any singularities over the range of interpolation. In addition, there should not be any pole in the vicinity of the complex plane near the portion of the real axis corresponding to the range of interpolation. If the function has singularities, using rational functions—that is, the quotient of two polynomials—instead is recommended [Press *et al.*].

In this chapter, three interpolation functions are discussed: the Lagrange interpolation polynomial, a diagonal rational function (Bulirsch-Stoer interpolation), and cubic spline. Furthermore, I show three different implementations, of the Lagrange interpolation polynomial: direct implementation of Lagrange's formula, Newton's algorithm, and Neville's algorithm. Figure 3.1 shows how the classes corresponding to the different interpolation methods described in this chapter are related to each other.

### 3.1.1 Interpolation Concepts and Examples

The *Lagrange interpolation polynomial* is the unique polynomial with minimum degree going through the sample points. The degree of the polynomial is equal to the number of supplied points minus 1. A *diagonal rational function* is the quotient of two polynomials where the degree of the polynomial in the numerator is at most equal to that of the denominator. *Cubic spline* uses piecewise interpolation with polynomials but limits the degree of each polynomial to 3 (hence the adjective *cubic*).

Before selecting an interpolation method, the user must investigate the validity of the interpolated function over the range of its intended use. Let us illustrate this remark with an example from high-energy physics that will describe the limitation of both methods exposed in this chapter.

Figure 3.2 shows sample points—indicated by crosses—representing correction to the energy measured within a gamma ray detector made of several densely packed crystals. The energy is plotted on a logarithmic scale. The correction is caused by the absorption of energy in the wrapping of each crystal. The sample points were computed using a simulation program,[2] each point requiring several hours of

2. This program—EGS, written by Ralph Nelson of the Stanford Linear Accelerator Center (SLAC)—simulates the absorption of electromagnetic showers inside matter. Besides being used in high-energy physics, EGS is also used in radiology to dimension detectors of PET scanners and other similar radiology equipment.

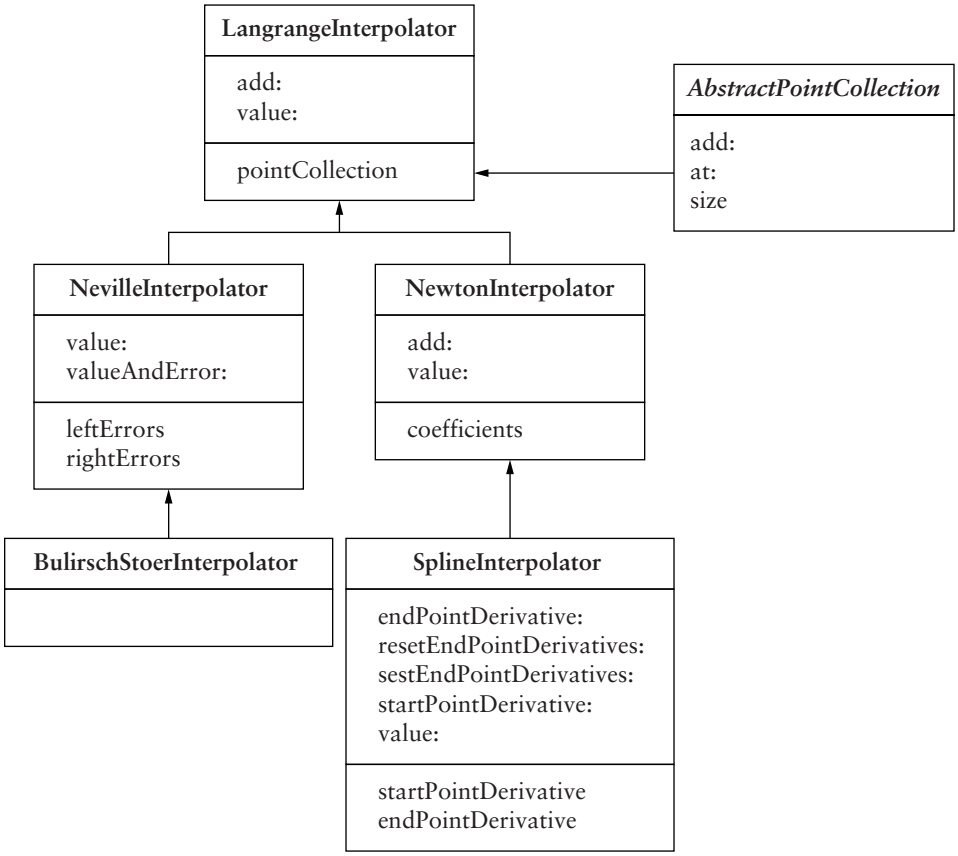


**FIG. 3.1** Class diagram for the interpolation classes

computing time. Interpolation over these points was therefore used to allow a quick computation of the correction at any energy. This is the main point of this example: the determination of each point was expensive in terms of computing time, but the function represented by these points is continuous enough to be interpolated. The simulation program yields results with good precision so that the resulting data are not subjected to fluctuation.

The gray thick line in Figure 3.2 shows the Lagrange interpolation polynomial obtained from the sample points. It readily shows limitations inherent to the use of interpolation polynomials. The reader can see that for values above 6.5—corresponding to an energy of 500 MeV—the interpolated function does not reproduce the curve corresponding to the sample points. In fact, above 4.0 (i.e., 50 MeV on the scale of Figure 3.2), the correction is expected to be a linear function of the logarithm of the energy.

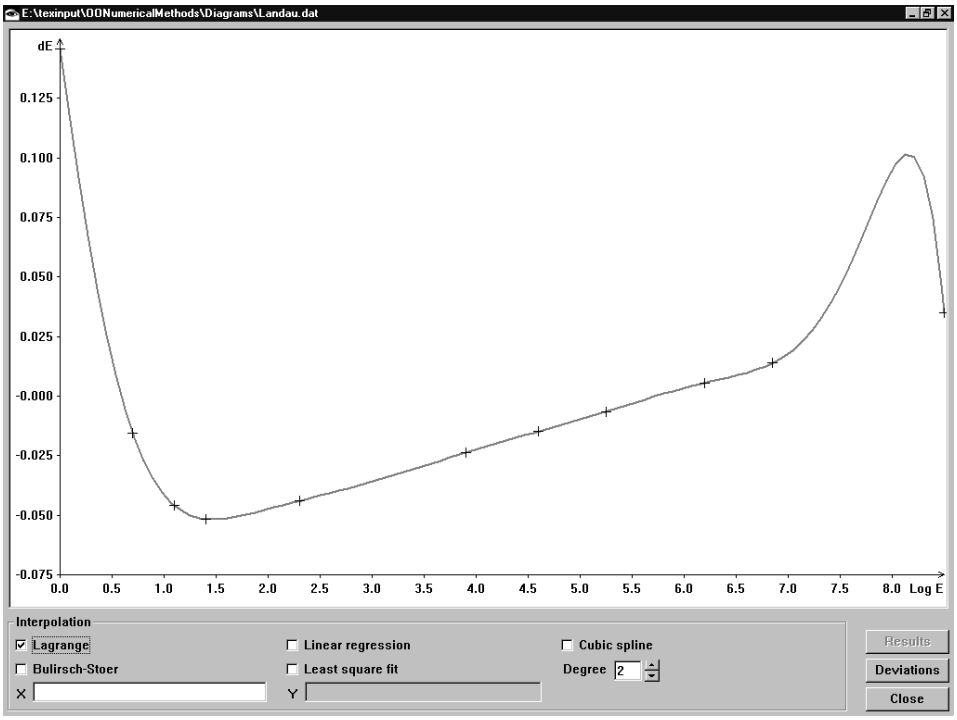


**FIG. 3.2** Example of interpolation with the Lagrange interpolation polynomial

Figure 3.3 shows a comparison between the Lagrange interpolation polynomial (gray thick line) and interpolation with a rational function (black dotted line) using the same sample points as in Figure 3.2. The reader can see that, in the high-energy region (above 4 on the scale of Figure 3.3), the rational function does a better job than the Lagrange polynomial. Between the first two points, however, the rational function fails to reproduce the expected behavior.

Figure 3.4 shows a comparison between the Lagrange interpolation polynomial (gray thick line) and cubic spline interpolation (black dotted line) using the same sample points as in Figure 3.2. The reader can see that, in the high-energy region (above 4 on the scale of Figure 3.3), cubic spline does a better job than the Lagrange polynomial. In fact, since the dependence is linear over that range, cubic spline reproduces the theoretical dependence exactly. In the low-energy region, however, cubic spline interpolation fails to reproduce the curvature of the theoretical function because of the limitation of the polynomial's degree.

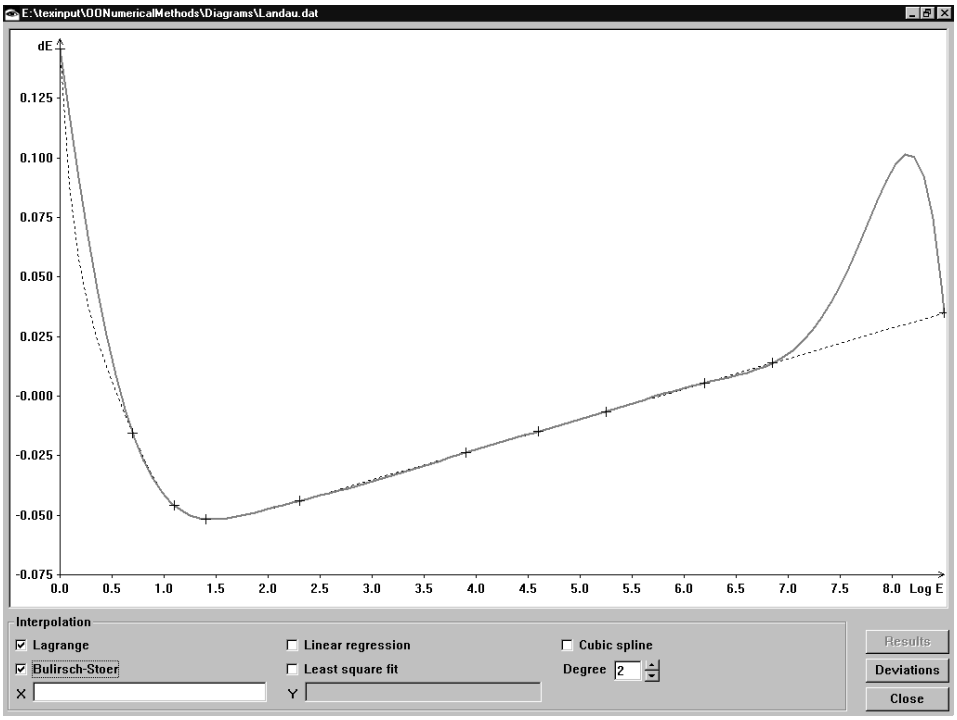


**FIG. 3.3** Comparison between Lagrange interpolation (gray line) and interpolation with a rational function (dotted line)

A final example shows a case in which interpolation should not be used. Here the sample points represent the dependence of the probability that a coin mechanism accepts a wrong coin as a function of an adjustable threshold. The computation of each point represents 5 to 10 minutes of computing time. In this case, however, the simulation was based on using experimental data. Contrary to the points of Figure 3.2, the points of Figure 3.5 are subjected to large fluctuations, because the sample points have been derived from measured data. Thus, interpolation does not work.

As in Figure 3.3, the gray thick line is the Lagrange interpolation polynomial, and the black dotted line is cubic spline. Clearly the Lagrange interpolation polynomial is not giving any reasonable interpolation. Cubic spline is not really better as it tries very hard to reproduce the fluctuations of the computed points. In this case, a polynomial fit (see Section ???) is the best choice: the thin black line shows the result of a fit with a third-degree polynomial. Another example of unstable interpolation is given in Section ??? (Figure ???).

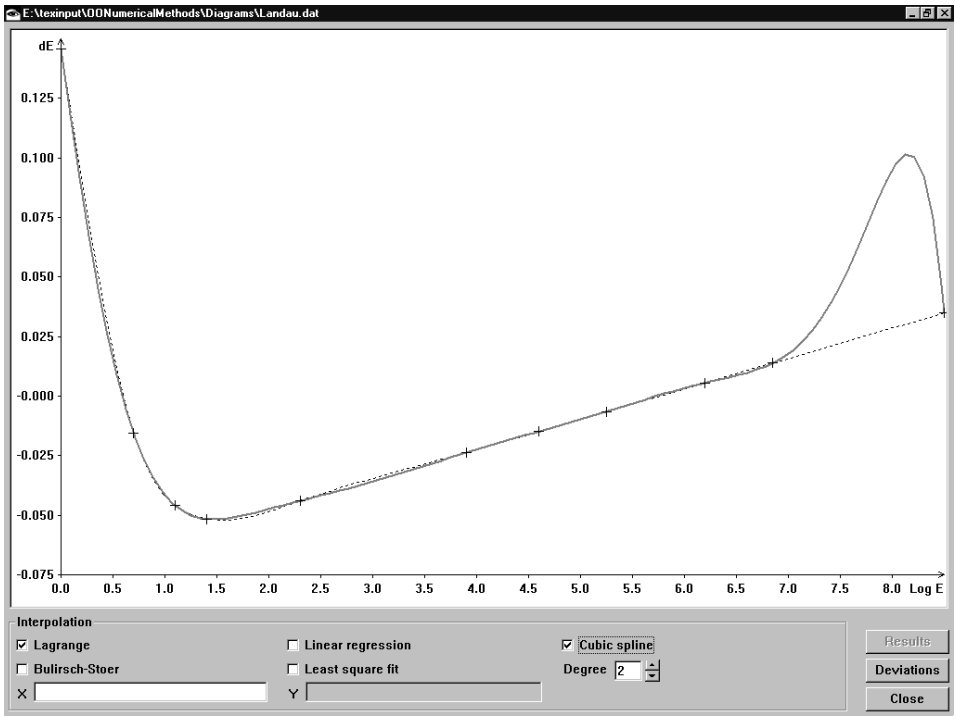


**FIG. 3.4** Comparison of Lagrange interpolation (gray line) and cubic spline (dotted line)

## 3.2 Lagrange Interpolation

Once you have verified that a Lagrange interpolation polynomial can be used to perform reliable interpolation over the sample points, you must choose among three algorithms to compute the Lagrange interpolation polynomial: direct Lagrange formula, Newton's algorithm, and Neville's algorithm.

Newton's algorithm stores intermediate values that only depend on the sample points. It is thus recommended, as it is the fastest method to interpolate several values over the same sample points. Newton's algorithm is the method of choice to compute a function from tabulated values.

Neville's algorithm gives an estimate of the numerical error obtained by the interpolation and can be used when such information is needed. Romberg integration, discussed in Section 6.4, uses Neville's method for that reason.

Let us assume a set of numbers $x_0, \ldots, x_n$ and the corresponding function's values $y_0, \ldots, y_n$. A unique polynomial $P_n(x)$ of degree $n$ exists such that $P_n(x_i) = y_i$ for all $i = 0, \ldots, n$. This polynomial is the Lagrange interpolation polynomial whose expression is given by [Knudth 2]:

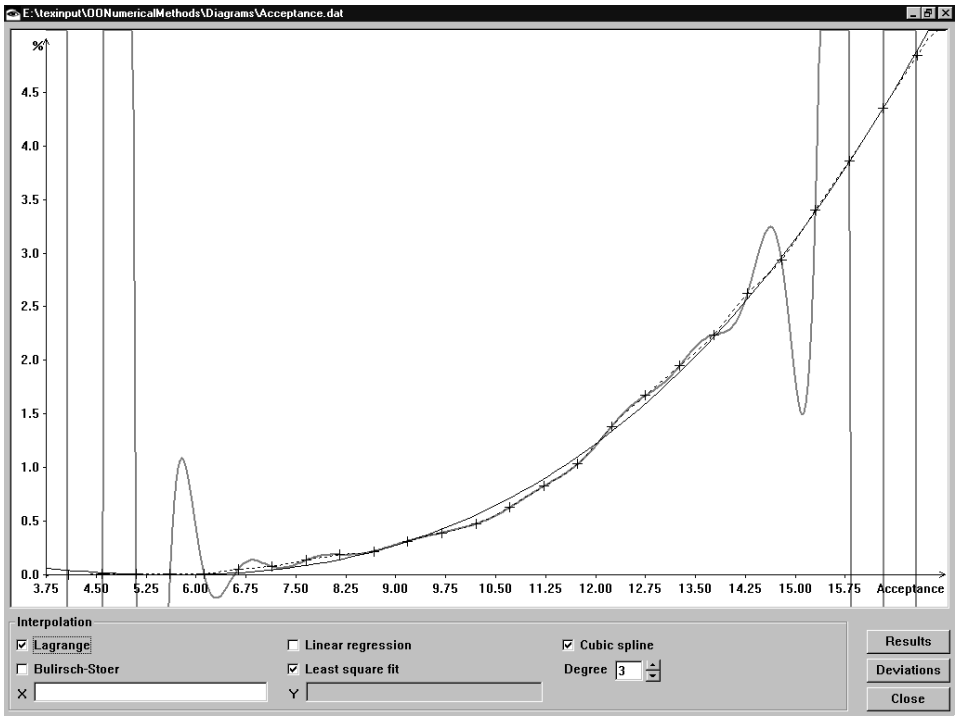


**FIG. 3.5** Example of misbehaving interpolation

$$P_n(x) = \sum_{i=0}^{n} \frac{\prod_{j \neq i} (x - x_j)}{\prod_{j \neq i} (x_i - x_j)} y_i. \tag{3.1}$$

For example, the Lagrange interpolation polynomial of degree 2 on three points is given by:

$$P_2(x) = \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} y_0 + \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_0 - x_2)} y_1 + \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)} y_2 \tag{3.2}$$

The computation of the polynomial occurs in the order of $\mathcal{O}(n^2)$ since it involves a double iteration. One can save the evaluation of a few products by rewriting equation 3.1 as:

$$P_n(x) = \prod_{i=0}^{n} (x - x_i) \sum_{i=0}^{n} \frac{y_i}{(x - x_i) \prod_{j \neq i} (x_i - x_j)}. \tag{3.3}$$

Of course, equation 3.3 cannot be evaluated at the points defining the interpolation. This is easily solved by returning the defining values as soon as one of the first products becomes 0 during the evaluation.

### 3.2.1 Lagrange Interpolation—Smalltalk Implementation

The object responsible to implement Lagrange interpolation is defined uniquely by the sample points over which the interpolation is performed. In addition, it should behave as a function. In other words, it should implement the behavior of a one-variable function as discussed in Section 2.1.1. For example, linear interpolation behaves as shown in Code Example 3.1.

**Code Example 3.1**

```
| interpolator |
interpolator := DhbLagrangeInterpolator points: ( Array with: 1 @ 2
                                                    with: 3 @ 1).
Interpolator value: 2.2
```

In this example, one creates a new instance of the class `DhbLagrangeInterpolator` by sending the message `points:` to the class `DhbLagrangeInterpolator` with the collection of sample points as argument. The newly created instance is stored in the variable `interpolator`. The next line shows how to compute an interpolated value.

The creation method `points:` takes as argument the collection of sample points. However, it could also accept any object implementing a subset of the methods of the class `Collection`—namely, the methods `size`, `at:`, and, if we want to be able to add new sample points, `add:`.

One can also spare the creation of an explicit collection object by implementing these collection methods directly in the Lagrange interpolation class. Now, one can also perform interpolation as given in Code Example 3.2.

**Code Example 3.2**

```
| interpolator deviation |
interpolator := DhbLagrangeInterpolator new.
1 to: 45 by: 2 do:
            [ :x | interpolator add: x @ (x degreesToRadians sin)].
deviation := (interpolator value: 8) -(8 degreesToRadians sin).
```

The code in this example creates an instance of the class `DhbLagrangeInterpolator` with an empty collection of sample points. It then adds sample points one by one directly into the interpolator object. Here the sample points are tabulated values of the sine function for odd-degree values between 1 and 45 degrees. The final line of the code compares the interpolated value with the correct one.

Listing 3.1 shows the full code of the class implementing the interface shown here.

The class `DhbLagrangeInterpolator` is implemented with a single-instance variable containing the collection of sample points. Each point contains a pair of values $(x_i, y_i)$ and is implemented with an object of the base class `Point` since an instance of `Point` can contain any type of object in its coordinates. There are two creation methods, `points:` and `new`, depending on whether the sample points are supplied as an explicit object or not. Each creation method calls in turn an initialization method, `initialize:` and `initialize`, respectively.

The method `points:` takes as argument the collection of the sample points. This object must implement the following methods of the class `Collection`: `size`, `at:`, and `add:`. If the class is created with the method `new`, an implicit collection object is created with the method `defaultSamplePoints`. This arrangement allows subclasses to select another type of collection if needed. The default collection behavior implemented by the class `DhbLagrangeInterpolator` is minimal, however. If there is a need for more flexible access to the collection of sample points, a proper collection object or a special purpose object should be used.

The interpolation itself is implemented within the single method `value:`. This method is unusually long for object-oriented programming standards. In this case, however, there is no compelling reason to split any portion of the algorithm into a separate method. Moreover, splitting the method would increase the computing time.

A final note should be made about the two methods `xPointAt:` and `yPointAt:`. In principle, there is no need for these methods as the value could be grabbed directly from the collection of points. If one needs to change the implementation of the point collection in a subclass, however, only these two methods need to be modified. Introducing this kind of construct can go a long way in program maintenance.

---

**Listing 3.1 Smalltalk implementation of the Lagrange interpolation**

| | |
|---|---|
| *Class* | `DhbLagrangeInterpolator` |
| *Subclass of* | `Object` |
| *Instance variable names:* | `pointCollection` |

***Class Methods***

**new**

```
^super new initialize
```

**points:** `aCollectionOfPoints`

```
^self new initialize: aCollectionOfPoints
```

### *Instance Methods*

**add:** aPoint

```
^pointCollection add: aPoint
```

**defaultSamplePoints**

```
^OrderedCollection new
```

**initialize**

```
^self initialize: self defaultSamplePoints
```

**initialize:** aCollectionOfPoints

```
pointCollection := aCollectionOfPoints.
^self
```

**value:** aNumber

```
| norm dx products answer size |
norm := 1.
size := pointCollection size.
products := Array new: size.
products atAllPut: 1.
1 to: size
    do: [ :n |
          dx := aNumber - ( self xPointAt: n).
          dx = 0
            ifTrue: [ ^( self yPointAt: n)].
          norm := norm * dx.
          1 to: size
            do: [ :m |
                  m = n
                    ifFalse:[ products at: m put: ( (( self
        xPointAt: m) - ( self xPointAt: n)) * ( products at: m))].
                ].
        ].
answer := 0.
1 to: size do:
    [ :n | answer := ( self yPointAt: n) / ( ( products at: n) *
                      ( aNumber - ( self xPointAt: n))) + answer].
^norm * answer
```

**xPointAt:** anInteger

```
^( pointCollection at: anInteger) x
```

**yPointAt:** `anInteger`

```
^( pointCollection at: anInteger) y
```

### 3.2.2 Lagrange Interpolation—Java Implementation

The object responsible to implement Lagrange interpolation is defined uniquely by the sample points over which the interpolation is performed. Unlike in Smalltalk, however, point objects cannot be used to hold the values because the Java class `Point` is only defined for integer values.

Thus, an interface called `PointSeries` was created to define the behavior of the object containing the sample points. This interface is shown in Listing 3.2.

**Listing 3.2 Java interface for point series**

```
package DhbInterpolation;

import DhbInterfaces.OneVariableFunction;
import DhbInterfaces.PointSeries;


// A LagrangeInterpolator can be used to interpolate values
between
// a series of 2-dimensional points. The interpolation function is
// the Langrange interpolation polynomial of a degree equal to the
// number of points in the series minus one.

// @author Didier H. Besset

public class LagrangeInterpolator implements OneVariableFunction
{

    // Points containing the values.

    protected PointSeries points;

// Constructor method.
// @param pts the series of points.
// @see PointSeries

public LagrangeInterpolator(PointSeries pts)
{
    points = pts;
}
```

```
    // Computes the interpolated y value for a given x value.
    // @param aNumber x value.
    // @return interpolated y value.

    public double value( double aNumber)
    {
        double norm = 1.0;
        int size = points.size();
        double products[] = new double[size];
        for ( int i = 0; i < size; i++)
            products[i] = 1;
        double dx;
        for ( int i = 0; i < size; i++)
        {
            dx = aNumber - points.xValueAt( i);
            if ( dx == 0 )
                return points.yValueAt(i);
            norm *= dx;
            for ( int j = 0; j < size; j++)
            {
                if ( i != j)
                    products[j] *= points.xValueAt(j)
                                                - points.xValueAt(i);
            }
        }
        double answer = 0.0;
        for ( int i = 0; i < size; i++)
            answer += points.yValueAt(i)
                        / (products[i] * (aNumber - points.xValueAt(i)));
        return norm * answer;
    }
    }
```

Listing 3.3 shows the class `Curve` implementing the minimum functionality of the interface `PointSeries`. This class is used in a variety of problems where handling geometrical points is required.

**Listing 3.3 A possible concrete implementation of the interface `PointSeries`**

```
package DhbScientificCurves;

import java.util.Vector;
import DhbInterfaces.PointSeries;

// A Curve is a series of points. A point is implemented as an array
```

```
// of two doubles. The points are stored in a vector so that points
// can be added or removed.

// @author Didier H. Besset

public class Curve implements PointSeries
{

    // Vector containing the points.

    protected Vector points;

// Constructor method. Initializes the vector.

public Curve()
{
    points = new Vector();
}

// Adds a point to the curve defined by its 2-dimensional coordinates.
// @param x double x-coordinate of the point
// @param y double y-coordinate of the point

public void addPoint( double x, double y)
{
    double point[] = new double[2];
    point[0] = x;
    point[1] = y;
    points.addElement( point);
}

// Removes the point at the specified index.
// @param int index of the point to remove

public void removePointAt( int index)
{
    points.removeElementAt( index);
}

// @return int the number of points in the curve.

public int size()
{
    return points.size();
}
```

```
    // @return double the x coordinate of the point at the given index.
    // @param int index the index of the point.

    public double xValueAt( int index)
    {
        return ((double[]) points.elementAt( index))[0];
    }

    // @return double the y coordinate of the point at the given index.
    // @param int index the index of the point.

    public double yValueAt( int index)
    {
        return ((double[]) points.elementAt( index))[1];
    }
    }
```

The object implementing Lagrange interpolation should implement the interface `OneVariableFunction` discussed in Section 2.1.1. This completes the public interface to the object. Thus, an example of linear interpolation behaves as shown in Code Example 3.3.

**Code Example 3.3**

```
DhbScientificCurves.Curve points =
        new DhbScientificCurves.Curve();
points.addPoint( 1, 2);
points.addPoint( 3, 1);
(new LagrangeInterpolator( points)).value( 2.2)
```

The first line creates the collection of sample points. The next two lines populate the collection with given sample points. The last line creates an instance of the class `LagrangeInterpolator` and evaluates its value at the desired point.

Code Example 3.4 shows how to tabulate a function.

**Code Example 3.4**

```
double radian = Math.PI / 180;
DhbScientificCurves.Curve points = new DhbScientificCurves.Curve();
for( int i = 1; i <= 45; i += 2)
       points.addPoint( i, Math.sin( i * radian));
LagrangeInterpolator interpolator =
                                  new LagrangeInterpolator( points);
double deviation =interpolator.value( 8) -
                        Math.sin( 8 * radian);
```

The code in Code Example 3.4 creates an instance of `Curve`, an object containing the sample points. This object is then populated with the values of the sine function for odd-degree values between 1 and 45 degrees. Then, an instance of the class `LagrangeInterpolator` is created on the sample points. The last line of the code computes the difference between the interpolated value and the exact value.

Listing 3.4 shows the full code of the class implementing the interface shown here.

The class `LagrangeInterpolator` is implemented with a single instance variable containing the sample points. Each point contains the pair of values $(x_i, y_i)$ defining the function. The object containing the sample points can be any object implementing the interface `PointSeries`. The single-constructor method takes the object containing the sample points as sole argument.

The interpolation itself is implemented within the single method `value:`. This method is unusually long by object-oriented programming standards. In this case, however, there is no compelling reason to split any portion of the algorithm into a separate method. Moreover, splitting the method would increase the computing time.

**Listing 3.4 Java implementation of the Lagrange interpolation**

```
package DhbInterpolation;

import DhbInterfaces.OneVariableFunction;
import DhbInterfaces.PointSeries;

// A LagrangeInterpolator can be used to interpolate values between
// a series of 2-dimensional points. The interpolation function is
// the Langrange interpolation polynomial of a degree equal to the
// number of points in the series minus one.

// @author Didier H. Besset

public class LagrangeInterpolator implements OneVariableFunction
{

    // Points containing the values.

    protected PointSeries points;

// Constructor method.
// @param pts the series of points.
// @see PointSeries

public LagrangeInterpolator(PointSeries pts)
{
```

```
        points = pts;
    }

    // Computes the interpolated y value for a given x value.
    // @param aNumber x value.
    // @return interpolated y value.

    public double value( double aNumber)
    {
        double norm = 1.0;
        int size = points.size();
        double products[] = new double[size];
        for ( int i = 0; i < size; i++)
            products[i] = 1;
        double dx;
        for ( int i = 0; i < size; i++)
        {
            dx = aNumber - points.xValueAt( i);
            if ( dx == 0 )
                return points.yValueAt(i);
            norm *= dx;
            for ( int j = 0; j < size; j++)
            {
                if ( i != j)
                    products[j] *= points.xValueAt(j)
                                                - points.xValueAt(i);
            }
        }
        double answer = 0.0;
        for ( int i = 0; i < size; i++)
            answer += points.yValueAt(i)
                        / (products[i] * (aNumber - points.xValueAt(i)));
        return norm * answer;
    }
    }
```

## 3.3 Newton Interpolation

If one must evaluate the Lagrange interpolation polynomial for several values, it is clear that the Lagrange's formula is not efficient. Indeed, a portion of the terms in the summation of equation 3.3 depends only on the sample points and does not depend on the value at which the polynomial is evaluated. Thus, one can speed up the evaluation of the polynomial if the invariant parts are computed once and stored.

If one writes the Lagrange interpolation polynomial using a generalized Horner expansion, one obtains the Newton's interpolation formula given by equation 3.4 [Knudth 2]:

$$P_n(x) = \alpha_0 + (x - x_0) \cdot [\alpha_1 + (x - x_1) \cdot [\cdots [\alpha_{n-1} + \alpha_n \cdot (x - x_1)]]] \tag{3.4}$$

The coefficients $\alpha_i$ are obtained by evaluating divided differences as follows:

$$\begin{cases} \Delta_i^0 = y_i \\ \Delta_i^k = \frac{\Delta_i^{k-1} - \Delta_{i-1}^{k-1}}{x_i - x_{i-k}} \quad \text{for } k = 1, \ldots, n \cdot \\ \alpha_i = \Delta_i^i \end{cases} \tag{3.5}$$

Once the coefficients $\alpha_i$ have been obtained, they can be stored in the object, and the generalized Horner expansion of equation 3.4 can be used.

The time to evaluate the full Newton's algorithm—that is, computing the coefficients and evaluating the generalized Horner expansion—is about twice the time needed to perform a direct Lagrange interpolation. The evaluation of the generalized Horner expansion alone, however, has an execution time of $\mathcal{O}(n)$ and is therefore much faster than the evaluation of a direct Lagrange interpolation, which goes as $\mathcal{O}(n^2)$. Thus, as soon as one needs to interpolate more than two points, Newton's algorithm is more efficient than direct Lagrange interpolation.

Note: that the implementations of Newton's interpolation algorithm are identical in both Smalltalk and Java. Thus, the reader can skip one of the two next subsections without losing anything.

### 3.3.1 Newton Interpolation—General Implementation

The object implementing Newton's interpolation algorithm is best implemented as a subclass of the class `DhbLagrangeInterpolator` because all methods used to handle the sample points can be reused. This also allows us to keep the interface identical. It has an additional instance variable needed to store the coefficients $\alpha_i$. Only four new methods are needed.

Since the client object can add new sample points at will, one cannot be sure when it is safe to compute the coefficients. Thus, computing the coefficients is done with lazy initialization. The method `value:` first checks whether the coefficients $\alpha_i$ have been computed. If not, the method `computeCoefficients` is called. Lazy initialization is a technique widely used in object-oriented programming whenever some value needs only be computed once.

The generalized Horner expansion is implemented in the method `value:`.

If a new sample point is added, the coefficients eventually stored in the object are no longer valid. Thus, the method `add:` first calls the method `resetCoefficients` and then calls the method `add:` of the superclass. The method `resetCoefficients` makes sure that the coefficients will be computed anew at the next evaluation of the interpolation polynomial. The method `resetCoefficients` has been implemented as a separate method so that the reset mechanism can be reused by any subclass.

Another reason to keep the method `resetCoefficients` separate is that it must also be called before doing an interpolation if the sample points have been modified directly by the client application after the last interpolation has been made. An alternative is to implement the OBSERVABLE/OBSERVER pattern so that resetting of the coefficients happens implicitly using events. However, since modifying the sample points between interpolation should only be a rare occasion when using Newton's algorithm[3] our proposed implementation is much simpler.

### 3.3.2 Newton Interpolation—Smalltalk Implementation

Listing 3.5 shows the complete implementation in Smalltalk. The class `NewtonInterpolator` is a subclass of class `LagrangeInterpolator`. Code Examples 3.1 and 3.2 can be directly applied to Newton interpolation after replacing the class name `DhbLagrangeInterpolator` with `DhbNewtonInterpolator`.

The generalized Horner expansion is implemented in the method `value:` using explicit indices. One could have used the method `inject:into:` as it was done for Horner's formula when evaluating polynomials. In this case, however, one must still keep track of the index to retrieve the sample point corresponding to each coefficient. Thus, one gains very little in compactness.

---

**Listing 3.5 Smalltalk implementation of the Newton interpolation**

| | |
|---|---|
| *Class* | `DhbNewtonInterpolator` |
| *Subclass of* | `DhbLagrangeInterpolator` |
| *Instance variable names:* | coefficients |

***Instance Methods***

**add:** `aPoint`

```
self resetCoefficients.
^super add: aPoint
```

**computeCoefficients**

```
| size k1 kn|
size := pointCollection size.
coefficients := ( 1 to: size) collect: [ :n | self yPointAt: n].
1 to: (size - 1)
    do: [ :n |
          size to: ( n + 1)  by: -1
```

3. If modification of the sample points is not a rare occasion, then Newton's algorithm has no advantage over direct Lagrange interpolation or Neville's algorithm. Those algorithms should be used instead of Newton's algorithm.

```
            do: [ :k |
                  k1 := k - 1.
                  kn := k - n.
                  coefficients at: k put: ( (( coefficients at: k)
                                          - ( coefficients at: k1))
                                          / ((self xPointAt: k) -
                                              (self xPointAt: kn))).
                ].
        ].
```

**resetCoefficients**

```
coefficients := nil.
```

**value:** aNumber

```
| answer size |
coefficients isNil
    ifTrue: [ self computeCoefficients].
size := coefficients size.
answer := coefficients at: size.
(size - 1) to: 1 by: -1
    do: [ :n | answer := answer * ( aNumber - (self xPointAt:
                                      n)) + ( coefficients at: n)].
^answer
```

### 3.3.3 Newton Interpolation—Java Implementation

Listing 3.6 shows the complete implementation in Java. The class NewtonInterpolator is a subclass of class LagrangeInterpolator. Code Examples 3.3 and 3.4 can be directly applied to Newton interpolation after replacing the class name LagrangeInterpolator with NewtonInterpolator.

Java has is an additional argument against implementing an OBSERVABLE/OBSERVER pattern to reset the coefficients automatically: an explicit finalization method is required to detach the link between OBSERVER and OBSERVABLE when the interpolator is no longer needed. Our simple implementation does not require finalization.

**Listing 3.6 Java implementation of the Newton interpolation**

```
package DhbInterpolation;

import DhbInterfaces.PointSeries;

// A NewtonInterpolator can be used to interpolate values between
// a series of 2-dimensional points. The interpolation function is
// the Langrange interpolation polynomial of a degree equal to
```

```
// the number of points in the series minus one. The coefficients
// of the polynomial are stored, speeding up interpolation for a
// series of values.

// @author Didier H. Besset

public class NewtonInterpolator extends LagrangeInterpolator
{

    // Polynomial coefficient (modified Horner expansion).

    protected double coefficients[];



// Constructor method.
// @param pts interfaces.PointSeries

public NewtonInterpolator( PointSeries pts) {
    super(pts);
}

// Computes the coefficients of the interpolation polynomial.

private void computeCoefficients()
{
    int size = points.size();
    int n;
    int k;
    int k1;
    int kn;
    coefficients = new double[size];
    for ( n = 0; n < size; n++)
        coefficients[n] = points.yValueAt(n);
    size -= 1;
    for ( n = 0; n < size; n++)
    {
        for ( k = size; k > n; k--)
        {
            k1 = k - 1;
            kn = k - (n + 1);
            coefficients[k] = ( coefficients[k] - coefficients[k1])
                                / ( points.xValueAt(k)
                                        - points.xValueAt(kn));
        }
    }
```

```
}

// Forces a new computation of the coefficients. This method must
be
// called whenever the series of points defining the interpolator
is
// modified.

public void resetCoefficients()
{
    coefficients = null;
}

// Computes the interpolated y value for a given x value.
// @param aNumber x value.
// @return interpolated y value.

public double value( double aNumber)
{
    if ( coefficients == null)
        computeCoefficients();
    int size = coefficients.length;
    double answer = coefficients[--size];
    while ( --size >= 0 )
        answer = answer * ( aNumber - points.xValueAt(size))
                                                + coefficients[size];
    return answer;
}
}
```

## 3.4 Neville Interpolation

Neville's algorithm uses a successive approximation approach implemented in practice by calculating divided differences recursively. The idea behind the algorithm is to compute the value of the interpolation's polynomials of all degrees between zero and $n$. This algorithm assumes that the sample points have been sorted in increasing order of abscissa.

Let $P_j^i(x)$ be the (partial) Lagrange interpolation polynomials of degree $i$ defined by the sets of values $x_j, \ldots, x_{j+i}$ and the corresponding function's values $y_j, \ldots, y_{j+i}$. From equation 3.1, one can derive the recurrence formula [Press *et al.*]:

$$P_j^i(x) = \frac{(x - x_{i+j})P_j^{i-1}(x) + (x_j - x)P_{j+1}^{i-1}(x)}{x_j - x_{i+j}} \quad \text{for } j < i. \tag{3.6}$$

The initial values $P_j^0(x)$ are simply $y_j$. The value of the final Lagrange polynomial is $P_0^n(x)$.

Neville's algorithm introduces the differences between the polynomials of various degrees. One defines:

$$\begin{cases} \Delta_{j,i}^{\text{left}}(x) & = P_j^i(x) - P_j^{i-1}(x) \\ \Delta_{j,i}^{\text{right}}(x) & = P_j^i(x) - P_{j+1}^{i-1}(x) \end{cases} \tag{3.7}$$

From this definition and equation 3.6, one derives a pair of recurrence formulas for the differences:

$$\begin{cases} \Delta_{j,i+1}^{\text{left}}(x) & = \dfrac{x_i - x}{x_j - x_{i+j+1}} \left[ \Delta_{j+1,i}^{\text{left}}(x) - \Delta_{j,i}^{\text{right}}(x) \right] \\ \Delta_{j,i+1}^{\text{right}} & = \dfrac{x_{i+j+1} - x}{x_j - x_{i+j+1}} \left[ \Delta_{j+1,i}^{\text{left}}(x) - \Delta_{j,i}^{\text{right}}(x) \right] \end{cases} \tag{3.8}$$

In practice, two arrays of differences—one for left and one for right—are allocated. Computation of each order is made within the same arrays. The differences of the last order can be interpreted as an estimation of the error made in replacing the function by the interpolation's polynomial.

Neville's algorithm is faster than the evaluation of direct Lagrange interpolation for a small number of points (fewer than about 7) see footnote 6 in Chapter 2. Therefore, a simple linear interpolation is best performed using Neville's algorithm. For a large number of points, it becomes significantly slower.

### 3.4.1 Neville Interpolation—General Implementation

The object implementing Neville interpolation's algorithm is best implemented as a subclass of the class `LagrangeInterpolator` since the methods used to handle the sample points can be reused. This method also allows us to keep the interface identical.

The new class has two additional instance variables used to store the finite differences $\Delta_{j,i}^{\text{left}}(x)$ and $\Delta_{j,i}^{\text{right}}(x)$ for all $j$. These instance variables are recycled for all $i$. Only a few additional methods are needed.

The method `valueAndError:` implementing Neville's algorithm returns an array with two elements: the first element is the interpolated value, and the second is the estimated error. The method `value:` calls the former method and returns only the interpolated value.

Unlike other interpolation algorithms, the method `valueAndError:` is broken into smaller methods because the mechanics of computing the finite differences will be reused in the Bulirsch-Stoer algorithm. The method `valueAndError:` begins by calling the method `initializeDifferences:` to populate the arrays containing the finite differences with their initial values. These arrays are created if this is the first time they are used with the current sample points, which prevents unnecessary mem-

ory allocation. Then, at each iteration the method `computeDifference:at:order:` computes the differences for the current order.

### 3.4.2 Neville Interpolation—Smalltalk Implementation

Listing 3.7 shows the implementation of Neville's algorithm in Smalltalk. The class `DhbNevilleInterpolator` is a subclass of class `DhbLagrangeInterpolator`. Code Examples 3.1 and 3.2 can be directly applied to Neville interpolation after replacing the class name `DhbLagrangeInterpolator` with `DhbNevilleInterpolator`. An example of interpolation using the returned estimated error is given in Section 6.4.2.

The method `defaultSamplePoints` overrides that of the superclass to return a sorted collection. Thus, each point added to the implicit collection is automatically sorted by increasing abscissa as required by Neville's algorithm.

---

**Listing 3.7 Smalltalk implementation of Neville's algorithm**

| | |
|---|---|
| *Class* | `DhbNevilleInterpolator` |
| *Subclass of* | `DhbLagrangeInterpolator` |
| *Instance variable names:* | `leftErrors rightErrors` |

*Instance Methods*

**computeDifference:** `aNumber` **at:** `anInteger1` **order:** `anInteger2`

```
| leftDist rightDist ratio |
leftDist := ( self xPointAt: anInteger1) - aNumber.
rightDist := (  self xPointAt: ( anInteger1 + anInteger2)) -
                                                    aNumber.
ratio := ( ( leftErrors at: ( anInteger1 + 1)) - ( rightErrors
                        at: anInteger1)) / ( leftDist - rightDist).
leftErrors at: anInteger1 put: ratio * leftDist.
rightErrors at: anInteger1 put: ratio * rightDist.
```

**defaultSamplePoints**

```
^SortedCollection sortBlock: [ :a :b | a x < b x]
```

**initializeDifferences:** `aNumber`

```
| size nearestIndex dist minDist |
size := pointCollection size.
leftErrors size = size
    ifFalse:[ leftErrors := Array new: size.
              rightErrors := Array new: size.
            ].
```

```
minDist := ( ( self xPointAt: 1) - aNumber) abs.
nearestIndex := 1.
leftErrors at: 1 put: ( self yPointAt: 1).
rightErrors at: 1 put: leftErrors first.
2 to: size do:
    [ :n |
      dist := ( ( self xPointAt: n) - aNumber) abs.
      dist < minDist
        ifTrue: [ dist = 0
                    ifTrue: [ ^n negated].
                  nearestIndex := n.
                  minDist := dist.
                ].
     leftErrors at: n put: ( self yPointAt: n).
     rightErrors at: n put: ( leftErrors at: n).
    ].
^nearestIndex
```

**value:** aNumber

```
^(self valueAndError: aNumber) first
```

**valueAndError:** aNumber

```
| size nearestIndex answer error |
nearestIndex := self initializeDifferences: aNumber.
nearestIndex < 0
    ifTrue: [ ^Array with: ( self yPointAt: nearestIndex negated)
                                                          with: 0].
answer := leftErrors at: nearestIndex.
nearestIndex := nearestIndex - 1.
size := pointCollection size.
1 to: ( size - 1) do:
    [ :m |
      1 to: ( size - m) do:
        [ :n | self computeDifference: aNumber at: n order: m].
      size - m > ( 2 * nearestIndex)
            ifTrue: [ error := leftErrors at: ( nearestIndex + 1)
                                                                ]
            ifFalse:[ error := rightErrors at: ( nearestIndex).
                        nearestIndex := nearestIndex - 1.
                    ].
      answer := answer + error.
    ].
^Array with: answer with: error abs
```

### 3.4.3 Neville Interpolation—Java Implementation

Listing 3.8 shows the implementation of Neville's algorithm in Java. The class `NevilleInterpolator` is a subclass of class `LagrangeInterpolator`. Code Examples 3.3 and 3.4 can be directly applied to Neville interpolation after replacing the class name `LagrangeInterpolator` with `NevilleInterpolator`. An example of interpolation using the returned estimated error is given in Section 6.4.3.

Note that since there is no sorted collection is Java, this class supposes that the calling application supplies the point sorted by ascending value of the abscissa.

**Listing 3.8 Java implementation of Neville interpolation**

```
package DhbInterpolation;


// NevilleInterpolator

// @author Didier H. Besset

public class NevilleInterpolator extends LagrangeInterpolator
{
    protected double[] leftErrors = null;
    protected double[] rightErrors = null;

// Constructor method.
// @param pts DhbInterfaces.PointSeries contains the points sampling
//           the function to interpolate.
// @exception java.lang.IllegalArgumentException points are not sorted
//           in increasing x values.

public NevilleInterpolator(DhbInterfaces.PointSeries pts)
{
    super(pts);
    for ( int i = 1; i < pts.size(); i++ )
    {
        if ( pts.xValueAt( i - 1) >= pts.xValueAt( i) )
            throw new IllegalArgumentException
                    ( "Points must be sorted in increasing x value");
    }
}

// @param m int    order of the difference
// @param n int    index of difference
// @param x double    argument

protected void computeNextDifference( int m, int n, double x)
```

```
{
    double leftDist = points.xValueAt(n) - x;
    double rightDist = points.xValueAt(n + m + 1) - x;
    double ratio = ( leftErrors[n+1] - rightErrors[n])
                                        / ( leftDist - rightDist);
    leftErrors[n] = ratio// leftDist;
    rightErrors[n] = ratio// rightDist;
}

// @return int

private int initializeDifferences( double x)
{
    int size = points.size();
    if ( leftErrors == null || leftErrors.length != size )
    {
        leftErrors = new double[ size];
        rightErrors = new double[ size];
    }
    double minDist = Math.abs( x - points.xValueAt(0));
    if ( minDist == 0 )
        return -1;
    int nearestIndex = 0;
    leftErrors[0] = points.yValueAt(0);
    rightErrors[0] = leftErrors[0];
    for ( int n = 1; n < size; n++)
    {
        double dist = Math.abs( x - points.xValueAt(n));
        if ( dist < minDist )
        {
            if ( dist == 0)
                return -n-1;
            minDist = dist;
            nearestIndex = n;
        }
        leftErrors[n] = points.yValueAt(n);
        rightErrors[n] = leftErrors[n];
    }
    return nearestIndex;
}

// @return double
// @param aNumber double

public double value( double aNumber)
{
```

```
    return valueAndError( aNumber)[0];
}

// @return double[]    an array with 2 elements:
//       [0] interpolated value, [1] estimated error
// @param x double

public double[] valueAndError( double x)
{
    double[] answer = new double[2];
    int nearestIndex = initializeDifferences( x);
    if ( nearestIndex < 0 )
    {
        answer[0] = points.yValueAt(-1-nearestIndex);
        answer[1] = 0;
        return answer;
    }
    int size = points.size();
    answer[0] = leftErrors[ nearestIndex--];
    double leftDist, rightDist, ratio;
    for ( int m = 0; m < size - 1; m++)
    {
        for ( int n = 0; n < size - 1 - m; n++)
        {
            computeNextDifference( m, n, x);
        }
        answer[1] = ( size - m > 2 * ( nearestIndex + 1) )
                        ? leftErrors[ nearestIndex + 1]
                        : rightErrors[ nearestIndex--];
        answer[0] += answer[1];
    }
    return answer;
}
}
```

## 3.5 Bulirsch-Stoer Interpolation

If the function to interpolate is known to have poles (i.e., a singularity in the complex plane.) in the vicinity of the real axis over the range of the sample points, a polynomial cannot do a good interpolation job [Press *et al.*]. In this case, it is better to use rational function—that is, a quotient of two polynomials as defined:

$$R(x) = \frac{P(x)}{Q(x)}. \tag{3.9}$$

The coefficients of both polynomials are only defined up to a common factor. Thus, if $p$ is the degree of polynomial $P(x)$ and $q$ is the degree of polynomial $Q(x)$, we must have the relation $p + q + 1 = n$, where $n$ is the number of sample points. This, of course, is not enough to restrict the variety of possible rational functions.

Bulirsch and Stoer have proposed an algorithm for a rational function where $p = \lfloor \frac{n-1}{2} \rfloor$. This means that $q$ is either equal to $p$ if the number of sample points is odd or equal to $p + 1$ if the number of sample points is even. Such a rational function is called a *diagonal rational function*. This restriction, of course, limits the type of function shapes that can be interpolated.

The Bulirsch-Stoer algorithm is constructed like Neville's algorithm: finite differences are constructed until all points have been taken into account.

Let $R_j^i(x)$ be the (partial) diagonal rational functions of order $i$ defined by the sets of values $x_j, \ldots, x_{j+i}$ and the corresponding function's values $y_j, \ldots, y_{j+i}$. As in the case of Neville's algorithm, one can establish a recurrence formula between functions of successive orders. We have [Press *et al.*]:

$$R_j^i(x) = R_{j+1}^{i-1}(x) + \frac{R_{j+1}^{i-1}(x) - R_j^{i-1}(x)}{\dfrac{x - x_j}{x - x_{i+j}} \left( 1 - \dfrac{R_{j+1}^{i-1}(x) - R_j^{i-1}(x)}{R_{j+1}^{i-1}(x) - R_{j+1}^{i-2}(x)} \right)} \quad \text{for } j < i. \tag{3.10}$$

The initial values $R_j^0(x)$ are simply $y_j$. The final rational function is $R_0^n(x)$.

As in Neville's algorithm, one introduces the differences between the functions of various orders. One defines:

$$\begin{cases} \Delta_{j,i}^{\text{left}}(x) & = R_j^i(x) - R_j^{i-1}(x) \\ \Delta_{j,i}^{\text{right}}(x) & = R_j^i(x) - R_{j+1}^{i-1}(x) \end{cases} \tag{3.11}$$

From this definition above and equation 3.10, one derives a pair of recurrence formulae for the differences:

$$\begin{cases} \Delta_{j,i+1}^{\text{left}}(x) = \dfrac{\dfrac{x - x_j}{x - x_{i+j+1}} \Delta_{j,i}^{\text{right}}(x) \left[ \Delta_{j+1,i}^{\text{left}}(x) - \Delta_{j,i}^{\text{right}}(x) \right]}{\dfrac{x - x_j}{x - x_{i+j+1}} \Delta_{j,i}^{\text{right}}(x) - \Delta_{j+1,i}^{\text{left}}(x)} \\ \Delta_{j,i}^{\text{right}}(x) = \dfrac{\Delta_{j+1,i}^{\text{left}}(x) \left[ \Delta_{j+1,i}^{\text{left}}(x) - \Delta_{j,i}^{\text{right}}(x) \right]}{\dfrac{x - x_j}{x - x_{i+j+1}} \Delta_{j,i}^{\text{right}}(x) - \Delta_{j+1,i}^{\text{left}}(x)} \end{cases} \tag{3.12}$$

As for Neville's algorithm, two arrays of differences—one for left and one for right—are allocated. Computation of each order is made within the same arrays. The differences of the last order can be interpreted as an estimation of the error made in replacing the function by the interpolating rational function. Given the many similarities with Neville's algorithm, many methods of that algorithm can be reused.

### 3.5.1 Bulirsch-Stoer Interpolation—General Implementation

The object implementing Bulirsch-Stoer interpolation's algorithm is best implemented as a subclass of the class `DhbNevilleInterpolator` since the methods used to manage the computation of the finite differences can be reused. The public interface is identical.

Only a single method—the one responsible for the evaluation of the finite differences at each order—must be implemented. All other methods of Neville's interpolation can be reused. This feature shows the great power of object-oriented approach. Code written in procedural language, in contrast, cannot be reused that easily. In [Press *et al.*] the two codes implementing Neville's and Bulirsch-Stoer interpolation are of comparable length; not surprisingly, they also have much in common.

### 3.5.2 Bulirsch-Stoer Interpolation—Smalltalk Implementation

Listing 3.9 shows the implementation of Bulirsch-Stoer interpolation in Smalltalk. The class `DhbBulirschStoerInterpolator` is a subclass of class `DhbNevilleInterpolator`. Code Examples 3.1 and 3.2 can be directly applied to Bulirsch-Stoer interpolation after replacing the class name `DhbLagrangeInterpolator` with `DhbBulirschStoerInterpolator`.

**Listing 3.9 Smalltalk implementation of Bulirsch-Stoer interpolation**

| | |
|---|---|
| *Class* | `DhbBulirschStoerInterpolator` |
| *Subclass of* | `DhbNevilleInterpolator` |

***Instance Methods***

**computeDifference:** `aNumber` **at:** `anInteger1`

**order:** `anInteger2`

```
| diff ratio |
ratio := ( ( self xPointAt: anInteger1) - aNumber) *
                                        ( rightErrors at: anInteger1)
                        / ( (  self xPointAt: ( anInteger1 +
                                            anInteger2)) - aNumber).
diff := ( ( leftErrors at: ( anInteger1 + 1))
                                         - ( rightErrors at: anInteger1))
                        / ( ratio - ( leftErrors at:
                                         ( anInteger1 + 1))).
rightErrors at: anInteger1 put: ( leftErrors at:
                                         ( anInteger1 + 1)) * diff.
leftErrors at: anInteger1 put: ratio * diff.
```

### 3.5.3 Bulirsch-Stoer Interpolation—Java Implementation

Listing 3.10 shows the implementation of Bulirsch-Stoer interpolation in Java. The class `BulirschStoerInterpolator` is a subclass of class `NevilleInterpolator`. Code Examples 3.3 and 3.4 can be directly applied to Bulirsch-Stoer interpolation after replacing the class name `LagrangeInterpolator` with `BulirschStoerInterpolator`.

**Listing 3.10 Java implementation of Bulirsch-Stoer interpolation**

```
package DhbInterpolation;


// Bulirsch-Stoer interpolation

// @author Didier H. Besset

public class BulirschStoerInterpolator extends NevilleInterpolator
{

// Constructor method.
// @param pts DhbInterfaces.PointSeries

public BulirschStoerInterpolator(DhbInterfaces.PointSeries pts) {
    super(pts);
}

// @param m int
// @param n int
// @param x double

}
protected void computeNextDifference( int m, int n, double x)
{
    double ratio = ( points.xValueAt(n) - x) * rightErrors[n]
                                / ( points.xValueAt(n + m + 1) - x);
    double diff = ( leftErrors[n+1] - rightErrors[n])
                                        / ( ratio - leftErrors[n+1]);
    if( Double.isNaN( diff) )
    {
        diff = 0;
    }
    rightErrors[n] = leftErrors[n+1] * diff;
    leftErrors[n] = ratio * diff;
}
}
```

## 3.6 Cubic Spline Interpolation

The Lagrange interpolation polynomial is defined globally over the set of given points and respective function's values. As we have seen in Figure 3.5 and to a lesser degree in Figure 3.4, Lagrange's interpolation polynomial can have large fluctuations between two adjacent points because the degree of the interpolating polynomial is not constrained.

One practical method for interpolating a set of function's value with a polynomial of constrained degree is to use cubic splines. A *cubic spline* is a third order polynomial constrained in its derivatives at the end points. A unique cubic spline is defined for each interval between two adjacent points. The interpolated function is required to be continuous up to the second derivative at each of the points.

Before the advent of computers, people were drawing smooth curves by sticking nails at the location of computed points and placing flat bands of metal between the nails. The bands were then used as rulers to draw the desired curve. These bands of metal were called *splines,* which is where the name of this interpolation algorithm comes from. The elasticity property of the splines corresponds to the continuity property of the cubic spline function.

The algorithm exposed hereafter assumes that the sample points have been sorted in increasing order of abscissa.

To derive the expression for the cubic spline, one first assumes that the second derivatives of the splines, $y_i''$, are known at each point. Then one writes the cubic spline between $x_{i-1}$ and $x_i$ in the following symmetrical form:

$$P_i(x) = y_{i-1}A_i(x) + y_iB_i(x) + y''_{i-1}C_i(x) + y''_iD_i(x), \tag{3.13}$$

where

$$\begin{cases} A_i(x) &= \frac{x_i - x}{x_i - x_{i-1}}, \\ B_i(x) &= \frac{x - x_{i-1}}{x_i - x_{i-1}}. \end{cases} \tag{3.14}$$

Using this definition, the first two terms in equation 3.13 represent the linear interpolation between the two points $x_{i-1}$ and $x_i$. Thus, the last two terms must vanish at $x_{i-1}$ and $x_i$. In addition, we must have by definition:

$$\begin{cases} \left.\frac{d^2P_i(x)}{dx^2}\right|_{x=x_{i-1}} = y''_{i-1}, \\ \left.\frac{d^2P_i(x)}{dx^2}\right|_{x=x_i} = y''_i. \end{cases} \tag{3.15}$$

One can rewrite the first equation in 3.15 as a differential equation for the function $C_i$ as a function of $A_i$. Similarly, the second equation is rewritten as a differential equation for the function $D_i$ as a function of $B_i$. This yields:

$$\begin{cases} C_i(x) = \dfrac{A_i(x)\left[A_i(x)^2 - 1\right]}{6}(x_i - x_{i-1})^2, \\ D_i(x) = \dfrac{B_i(x)\left[B_i(x)^2 - 1\right]}{6}(x_i - x_{i-1})^2, \end{cases} \tag{3.16}$$

Finally, one must use the fact that the first derivatives of each spline must be equal at each end point of the interval—that is,

$$\frac{dP_i(x)}{dx} = \frac{dP_{i+1}(x)}{dx}. \tag{3.17}$$

This yields the equations for the second derivatives $y_i''$:

$$\frac{x_{i+1} - x_i}{6}y''_{i+1} + \frac{x_{i+1} - x_{i-1}}{6}y''_i + \frac{x_i - x_{i-1}}{6}y''_{i-1} = \frac{y_{i+1} - y_i}{x_{i+1} - x_i} - \frac{y_i - y_{i-1}}{x_i - x_{i-1}}. \tag{3.18}$$

There are $n - 1$ equations for the $n$ unknowns $y_i''$. We are thus missing two additional equations to obtain a unique solution. Two ways of determination are possible:

1. The first method is the so-called *natural cubic spline* for which one sets $y_0'' = y_n'' = 0$. This means that the spline is flat at the end points.
2. The second method is called *constrained cubic spline*. In this case, the first derivatives of the function at $x_0$ and $x_n$, $y_0'$ and $y_n'$, are set to given values.

In the case of constrained cubic spline, one can obtain two additional equations by evaluating the derivatives of equation 3.13 at $x_0$ and $x_n$:

$$\begin{cases} \dfrac{3A_1(x)^2 - 1}{6}(x_1 - x_0)y_0'' - \dfrac{3B_1(x)^2 - 1}{6}(x_1 - x_0)y_1'' = y_0' - \dfrac{y_1 - y_0}{x_1 - x_0}, \\ \dfrac{3A_n(x)^2 - 1}{6}(x_n - x_{n-1})y_n'' - \dfrac{3B_n(x)^2 - 1}{6}(x_n - x_{n-1})y_{n-1}'' = y_n' - \dfrac{y_n - y_{n-1}}{x_n - x_{n-1}}. \end{cases} \tag{3.19}$$

The choice between natural or constrained spline can be made independently at each end point.

One solves the system of equations 3.18 and possibly 3.6 using direct Gaussian elimination and back substitution (see Section ???). Because the corresponding matrix is tridiagonal, each pivoting step only involves one operation. Thus, resorting to a general algorithm for solving a system of linear equations is not necessary.

### 3.6.1 Cubic spline Interpolation—General Implementation

In both languages, the object implementing cubic spline interpolation is a subclass of the Newton interpolator. The reader might be surprised by this choice since, mathematically speaking, these two objects do not have anything in common. However,

from the behavioral point of view, they are quite similar. As for Newton interpolation, cubic spline interpolation first needs to compute a series of coefficients—namely, the second derivatives—that only depends on the sample points. This calculation only needs to be performed once. Then the evaluation of the function can be done using equations 3.13, 3.14, and 3.16. Finally, as for the Newton interpolator, any modification of the points requires a new computation of the coefficients. The behavior can be reused from the class `NewtonInterpolator`.

The second derivatives needed by the algorithm are stored in the variable used to store the coefficients of Newton's algorithm.

The class `SplineInterpolator` has two additional instance variables needed to store the end point derivatives $y'_0$ and $y'_n$. Corresponding methods needed to set or reset these values are implemented. If the value of $y'_0$ or $y'_n$ is changed, then the coefficients must be reset.

Natural or constrained cubic spline is flagged independently at each point by testing whether the corresponding end point derivative has been supplied. The second derivatives are computed used lazy initialization by the method `computeSecondDerivatives`.

### 3.6.2 Cubic spline Interpolation—Smalltalk Implementation

Listing 3.11 shows the implementation of cubic spline interpolation in Smalltalk. The class `DhbSplineInterpolator` is a subclass of class `DhbNewtonInterpolator`. Code Examples 3.1 and 3.2 can be directly applied to cubic spline interpolation after replacing the class name `DhbLagrangeInterpolator` with `DhbSplineInterpolator`.

If the end point derivative is `nil`, the corresponding end point is treated as a natural spline.

The method `defaultSamplePoints` overrides that of the superclass to create a sorted collection. Thus, as each point is added to the implicit collection, the collection of sample points remains in increasing order of abscissa as required by the cubic spline algorithm.

---

**Listing 3.11 Smalltalk implementation of cubic spline interpolation**

| | |
|---|---|
| *Class* | `DhbSplineInterpolator` |
| *Subclass of* | `DhbNewtonInterpolator` |
| *Instance variable names:* | `startPointDerivative endPointDerivative` |

***Instance Methods***

**computeSecondDerivatives**

```
| size u w s dx inv2dx |
size := pointCollection size.
```

```
coefficients := Array new: size.
u := Array new: size - 1.
startPointDerivative isNil
    ifTrue:
        [coefficients at: 1 put: 0.
        u at: 1 put: 0]
    ifFalse:
        [coefficients at: 1 put: -1 / 2.
        s := 1 / (( self xPointAt: 2) x - ( self xPointAt: 1) x).
        u at: 1
            put: 3 * s
                    * (s * (( self yPointAt: size) -
                                    ( self yPointAt: size - 1))
                            - startPointDerivative)].
2 to: size - 1
    do:
        [:n |
        dx := (self xPointAt: n) - (self xPointAt: ( n - 1)).
        inv2dx := 1 / (( self xPointAt: n + 1) -
                                    (self xPointAt: n - 1)).
        s := dx * inv2dx.
        w := 1 / (s * (coefficients at: n - 1) + 2).
        coefficients at: n put: (s - 1) * w.
        u at: n
            put: ((((  self yPointAt: n + 1) -
                                    ( self yPointAt: n))
                    / (( self xPointAt: n + 1) -
                                    ( self xPointAt: n))
                        - ((( self yPointAt: n) -
                                    ( self yPointAt: n - 1)) / dx)) * 6
                    * inv2dx - ((u at: n - 1) * s))* w].
endPointDerivative isNil
    ifTrue: [coefficients at: size put: 0]
    ifFalse:
        [w := 1 / 2.
        s := 1 / ((self xPointAt:  size) -
                                    (self xPointAt: ( size - 1))).
        u at: 1
            put: 3 * s * (endPointDerivative
                            - (s * (self yPointAt: size) -
                                   (self yPointAt: size - 1))).
        coefficients at: size
            put: s - (w * (u at: size - 1) /
                            ((coefficients at: size - 1) * w + 1))].
size - 1 to: 1
    by: -1
```

```
    do:
        [:n |
        coefficients at: n
            put: (coefficients at: n) * (coefficients at: n + 1)
                                                    + (u at: n)]
```

**defaultSamplePoints**

```
^SortedCollection sortBlock: [ :a :b | a x < b x]
```

**endPointDerivative:** aNumber

```
endPointDerivative := aNumber.
self resetCoefficients.
```

**resetEndPointDerivatives**

```
self setEndPointDerivatives: ( Array new: 2).
```

**setEndPointDerivatives:** anArray

```
startPointDerivative := anArray at: 1.
endPointDerivative := anArray at: 2.
self resetCoefficients.
```

**startPointDerivative:** aNumber

```
startPointDerivative := aNumber.
self resetCoefficients.
```

**value:** aNumber

```
| answer n1 n2 n step a b |
coefficients isNil ifTrue: [self computeSecondDerivatives].
n2 := pointCollection size.
n1 := 1.
[n2 - n1 > 1] whileTrue:
        [n := (n1 + n2) // 2.
        (self xPointAt:  n) > aNumber ifTrue: [n2 := n]
                                    ifFalse: [n1 := n]].
step := (self xPointAt: n2) - (self xPointAt: n1).
a := ((self xPointAt: n2) - aNumber) / step.
b := (aNumber - (self xPointAt: n1)) / step.
^a * (self yPointAt: n1) + (b * (self yPointAt: n2))
    + ((a * (a squared - 1) * (coefficients at: n1)
            + (b * (b squared - 1) * (coefficients at: n2))) *
                                                    step squared / 6)
```

### 3.6.3 Cubic spline Interpolation—Java Implementation

Listing 3.12 shows the implementation of cubic spline interpolation in Java. The class `SplineInterpolator` is a subclass of class `NewtonInterpolator`. Code Examples 3.3 and 3.4 can be directly applied to cubic spline interpolation after replacing the class name `LagrangeInterpolator` with `SplineInterpolator`.

The fact that an end point derivative has not been supplied is flagged with the special Java value `Double.NaN`. Since there is no sorted collection is Java, this class supposes that the calling application supplies the point sorted by ascending value of the abscissa.

**Listing 3.12 Java implementation of cubic spline interpolation**

```
package DhbInterpolation;

import DhbInterfaces.OneVariableFunction;
import DhbInterfaces.PointSeries;

// A SplineInterpolator can be used to interpolate values between
// a series of 2-dimensional points. The interpolation function is
// a cubic spline with first derivatives defined at the end points.
// If the first derivatives are not defined for the end points,
// a so-called natural spline is used with second derivatives at
// the end points set to zero.

// @author Didier H. Besset

public class SplineInterpolator extends NewtonInterpolator
                        implements DhbInterfaces.OneVariableFunction
{

    // First derivative at first point.

    private double startPointDerivative = Double.NaN;

    // First derivative at last point.

    private double endPointDerivative = Double.NaN;

// This method creates a new instance of a spline interpolator over
// a given set of points. The points must be sorted by strictly
// increasing x values.
// @param pts DhbInterfaces.PointSeries contains the points sampling
//            the function to interpolate.
// @exception java.lang.IllegalArgumentException points are not sorted
//            in increasing x values.
```

```
public SplineInterpolator ( PointSeries pts)
                                    throws IllegalArgumentException
{
    super( pts);
    for ( int i = 1; i < pts.size(); i++ )
    {
        if ( pts.xValueAt( i - 1) >= pts.xValueAt( i) )
            throw new IllegalArgumentException
                    ( "Points must be sorted in increasing x value");
    }
}
private void computeSecondDerivatives( )
{
    int n = points.size();
    double w, s;
    double[] u = new double[ n - 1];
    coefficients = new double[ n];
    if ( Double.isNaN( startPointDerivative) )
        coefficients[0] = u[0] = 0;
    else
    {
        coefficients[0] = -0.5;
        u[0] = 3.0 / ( points.xValueAt( 1) - points.xValueAt( 0))
                    * ( ( points.yValueAt( 1) - points.yValueAt( 0))
                        / ( points.xValueAt( 1) - points.xValueAt( 0))
                                - startPointDerivative);
    }
    for ( int i = 1; i < n - 1; i ++ )
    {
        double invStep2 = 1 / ( points.xValueAt( i + 1)
                                        - points.xValueAt( i - 1));
        s = ( points.xValueAt( i) - points.xValueAt( i - 1)) * invStep2;
        w = 1 / ( s * coefficients[ i - 1] + 2);
        coefficients[ i] = ( s - 1) * w;
        u[i] = ( 6 * invStep2 * (
                    ( points.yValueAt( i + 1) - points.yValueAt( i))
                    / ( points.xValueAt( i + 1) - points.xValueAt( i))
                    - ( points.yValueAt( i) - points.yValueAt( i - 1))
                    / ( points.xValueAt( i) - points.xValueAt( i - 1))
                        ) - s * u[i - 1]) * w;
    }
    if ( Double.isNaN( endPointDerivative) )
        w = s = 0;
    else
    {
        w = -0.5;
```

```
            s = 3.0 / ( points.xValueAt( n - 1) - points.xValueAt( n - 2))
                        * ( endPointDerivative - ( points.yValueAt( n - 1
                            ) - points.yValueAt( n - 2))
                        / ( points.xValueAt( n - 1) -
                                                points.xValueAt( n - 2)));
        }
        coefficients[ n - 1] = ( s - w * u[ n - 2])
                                    / ( w * coefficients[ n - 2] + 1);
        for ( int i = n - 2; i >= 0; i--)
            coefficients[i] = coefficients[i] * coefficients[i + 1] + u[i];
        return;
    }

    // Computes the interpolated y value for a given x value.
    // @param aNumber x value.
    // @return interpolated y value.

    public double value(double x)
    {
        if ( coefficients == null )
            computeSecondDerivatives();
        int n1 = 0;
        int n2 = points.size() - 1;
        while ( n2 - n1 > 1 )
        {
            int n = (n1 + n2) / 2;
            if ( points.xValueAt( n) > x )
                n2 = n;
            else
                n1 = n;
        }
        double step = points.xValueAt( n2) - points.xValueAt( n1);
        double a = ( points.xValueAt( n2) - x) / step;
        double b = ( x - points.xValueAt( n1)) / step;
        return a * points.yValueAt( n1) + b * points.yValueAt( n2)
                   + ( a * ( a * a - 1) * coefficients[n1]
                           + b * ( b * b - 1) * coefficients[n2])
                                                    * step * step / 6;
    }
    }
```

## 3.7 Which Method to Choose?

At this point, some readers might experience some difficulty in choosing among the many interpolation algorithms discussed in this book. There are indeed many ways

to skin a cat. Selecting a method depends on what the user intends to do with the data.

First, the reader should be reminded that Lagrange interpolation, Newton interpolation, and Neville's algorithm are different alorithms computing the values of the same function—namely, the Lagrange interpolation polynomial. In other words, the interpolated value resulting from each of the three algorithms is the same (up to rounding errors, of course).

The Lagrange interpolation polynomial can be subject to strong variations (if not wild in some cases; see Figure 3.5 e.g.) if the sampling points are not smooth enough. A cubic spline may depart from the desired function if the derivatives on the end points are not constrained to proper values. A rational function can do a good job in cases in which polynomials have problems. To conclude, let me offer some rules of thumb to select the best interpolation method based on my personal experience.

If the function to interpolate is not smooth enough, which may be the case when not enough sampling points are available, a cubic spline is preferable to the Lagrange interpolation polynomial. Cubic splines are traditionally used in curve-drawing programs. Once the second derivatives have been computed, evaluation time is of the order of $\mathcal{O}(n)$. The limitation[4] imposed on the curvature when using a third order polynomial must be kept in mind.

If the Lagrange interpolation polynomial is used to evaluate quickly a tabulated[5] function, Newton interpolation is the algorithm of choice. As for cubic spline interpolation, the evaluation time is of the order of $\mathcal{O}(n)$ once the coefficients have been computed.

Neville's algorithm is the only choice if an estimate of error is needed in addition to the interpolated value. The evaluation time of the algorithm is of the order of $\mathcal{O}(n^2)$.

Lagrange interpolation can be used for occasional interpolation or when the values over which interpolation is made are changing at each interpolation. The evaluation time of the algorithm is of the order of $\mathcal{O}(n^2)$. Lagrange interpolation is slightly slower than Neville's algorithm as soon as the number of points is larger[6] than 3. However, Neville's algorithm needs to allocate more memory. Depending on the operating system and the amount of available memory, the exact place where Lagrange interpolation becomes slower than Neville's algorithm is likely to change.

If the function is smooth but a Lagrange polynomial is not reproducing the function in a proper way, a rational function can be tried using Bulirsch-Stoer interpolation.

---

4. The curvature of a cubic spline is somewhat limited. What happens is that the curvature and the slope (first derivative) are strongly coupled. As a consequence, a cubic spline gives a smooth approximation to the interpolated points.

5. A *tabulated function* is a function that has been computed at a finite number of its argument.

6. Such a number is strongly dependent on the operating system and virtual machine. Thus, the reader should check this number him or herself.

**TABLE 3.1** Recommended polynomial interpolation algorithms

| Feature | Recommended algorithm |
|---|---|
| Error estimate desired | Neville |
| Couple of sample points | Lagrange |
| Medium to large number of sample points | Neville |
| Many evaluations on fixed sample | Newton |
| Keep curvature under constraint | Cubic spline |
| Function hard to reproduce | Bulirsch-Stoer |

Table 3.1 shows a summary of the discussion. If you are in doubt, I recommend that you make a test first for accuracy and then for speed of execution. Drawing a graph such as those in the figures presented in this chapter is quite helpful to get a proper feeling about the possibility offered by various interpolation algorithms on a given set of sample points. If Lagrange interpolation, Bulirsch-Stoer, or cubic spline is not doing a good job at interpolating the sample points, you should consider using curve fitting (see Chapter ???) with an ad hoc function.

CHAPTER 4

# Iterative Algorithms

*Cent fois sur le métier remettez votre ouvrage.*[1]

—Nicolas Boileau

When a mathematical function cannot be approximated with a clever expression, such as the Lanczos formula introduced in Section 2.4.1, one must resort to computing that function using the integral, the recurrence formula, or the series expansion. All these algorithms have one central feature in common: the repetition of the same computation until some convergence criteria is met. Such repetitive computation is called *iteration*.

Figure 4.1 shows the class diagram of the classes considered in this chapter. In this chapter, I first discuss the implementation of a general-purpose iterative process. Then, I describe a generalization for finding a numerical result. Other chapters discuss examples of subclassing of these classes to implement specific algorithms.

Iteration is used to find the solution to a wide variety of problems other than just function evaluation. Finding the location where a function is zero or reached a maximum or a minimum is another example. Some data mining algorithms also use iteration to find a solution (see Section ???).

## 4.1 Successive Approximations

A general-purpose iterative process can be decomposed in three main steps:

1. a setup phase
2. an iteration phase until the result is acceptable
3. a cleanup phase

These steps are translated schematically in the flow diagram shown in Figure 4.2.

1. Take back your work to the loom a hundred times.

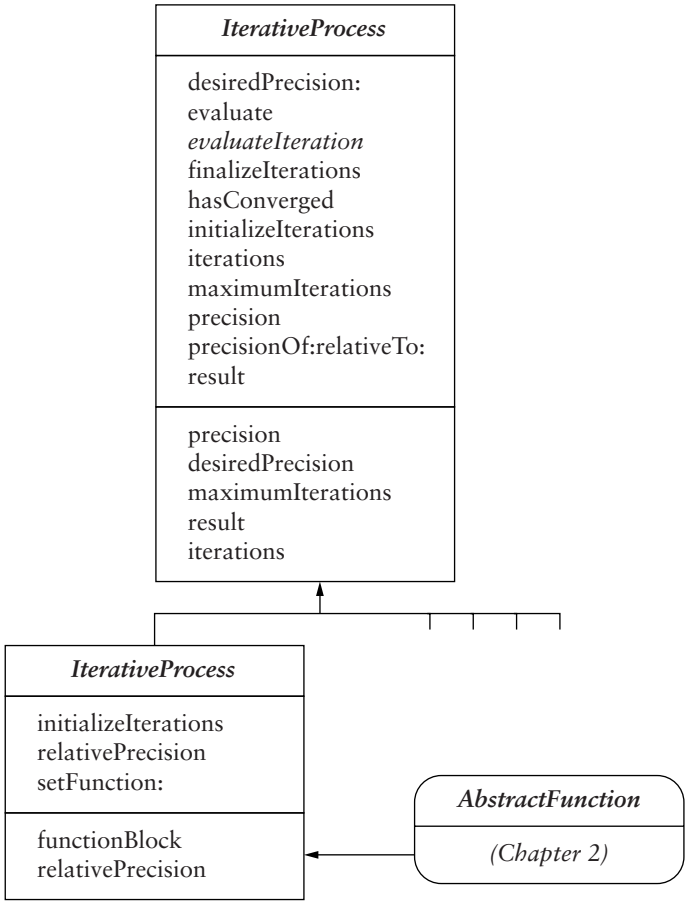


**FIG. 4.1** Class diagram for iterative process classes

The setup phase allows determining constant parameters used by the subsequent computations. Often a first estimation of the solution is defined at this time. In any case, an object representing the approximate solution is constructed. Depending on the complexity of the problem, a class will explicitly represent the solution object. Otherwise, the solution shall be described by a few instance variables of simple types (numbers and arrays).

After the setup phase, the iterative process proper is started. A transformation is applied to the solution object to obtain a new object. This process is repeated unless the solution object resulting from the last transformation can be considered close enough to the sought solution.

During the cleanup phase, resources used by the iterative process must be released. In some cases, additional results may be derived before leaving the algorithm.

Let us now explicitly examine each of the three stages of the algorithm.

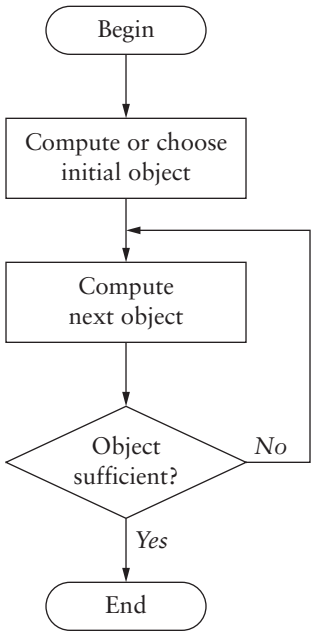


**FIG. 4.2** Successive approximation algorithm

The step computing or choosing an initial object is strongly dependent on the nature of the problem to be solved. In some methods, a good estimate of the solution can be computed from the data, in others using, randomly generated objects yields good results. Finally, one can also ask the application's user for directions. In many cases, this step is also used to initialize parameters needed by the algorithm.

The step computing the next object contains the essence of the algorithm. In general, a new object is generated based on the algorithm's history.

The step deciding whether an object is sufficiently close to the sought solution is more general. If the algorithm is capable of estimating the precision of the solution—that is, how close the current object is located from the exact solution—one can decide to stop the algorithm by comparing the precision to a desired value. This is not always the case, however. Some algorithms (e.g., genetic algorithms) do not have a criterion for stopping.

Whether or not a well-defined stopping criterion exists, the algorithm must be prevented from taking an arbitrary large amount of time. Thus, the object implementing an iterative process ought to keep track of the number of iterations and interrupt the algorithm if the number of iterations becomes larger than a given number.

Now we can add some details to the algorithm (the new details are shown in Figure 4.3). This schema allows us to determine the structure of a general object implementing the iterative process. It will be implemented as an abstract class. An

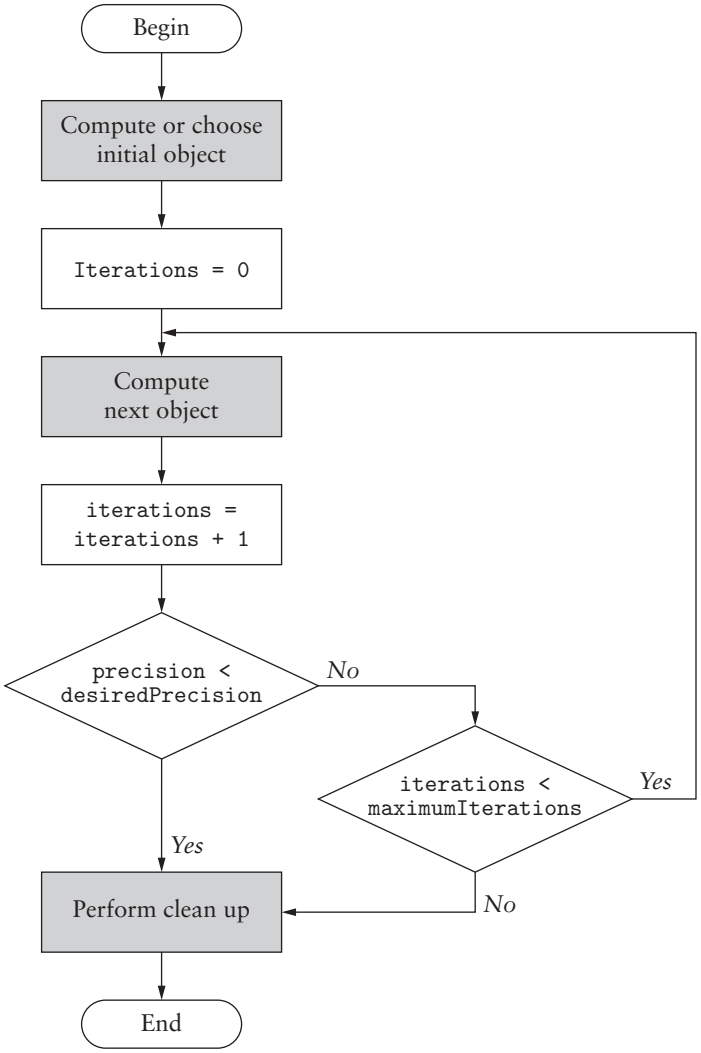


**FIG. 4.3** Detailed algorithm for successive approximations

*abstract* class is a class that does not have object instances. An object implementing a specific algorithm is an instance of a particular subclass of the abstract class.

The gray boxes in Figure 4.3 represent the methods, that must be implemented explicitly by the subclass. The abstract class calls them. However, the exact implementation of these methods is not defined at this stage. Such methods are called *hook* methods.

Using this architecture, the abstract class is able to implement the iterative process without any deep knowledge of the algorithm. A specific algorithm is implemented as a subclass of the abstract class.

Let us call `IterativeProcess` the class of the abstract object. The class `IterativeProcess` needs the following instance variables:

`iterations` Keeps track of the number of iterations—that is, the number of successive approximations

`maximumIterations` Maximum number of allowed iterations

`desiredPrecision` The precision to attain—that is, how close to the solution the solution object should be when the algorithm is terminated

`precision` The precision Achieved by the process, its value is updated after each iteration, and it is used to decide when to stop.

The methods of the class `IterativeProcess` are shown in Figure 4.4 in correspondence with the general execution flow shown in Figure 4.3. The two methods `initializeIterations` and `finalizeIterations` should be implemented by the subclass, but the abstract class provides a default behavior: doing nothing. The method `evaluateIteration` must be implemented by the subclass.

Since the precision of the last iteration is kept in an instance variable, the method `hasConverged` can be called at any time after evaluation, thus providing a way for client classes to check whether the evaluation has converged.

### 4.1.1 Iterative Process—Smalltalk Implementation

Even though we are dealing for the moment with an abstract class, we are able to present a scenario of use illustrating the public interface to the class. Code Example 4.1 shows what a basic utilization of an iterative process object would look like.

**Code Example 4.1**

```
| iterativeProcess result |
iterativeProcess := <a subclass of DhbIterativeProcess> new.
result := iterativeProcess evaluate.
iterativeProcess hasConverged
ifFalse:[ <special case processing>].
```

The first statement creates an object to handle the iterative process. The second one performs the process and retrieves the result, whatever it is. The final statement checks for convergence.

To give the user a possibility to have more control, one can extend the public interface of the object to allow defining the parameters of the iterative process: the desired precision and the maximum number of iterations. In addition, the user may want to know the precision of the attained result and the number of iterations needed to obtain the result. Code Example 4.2 shows an example of use for all public

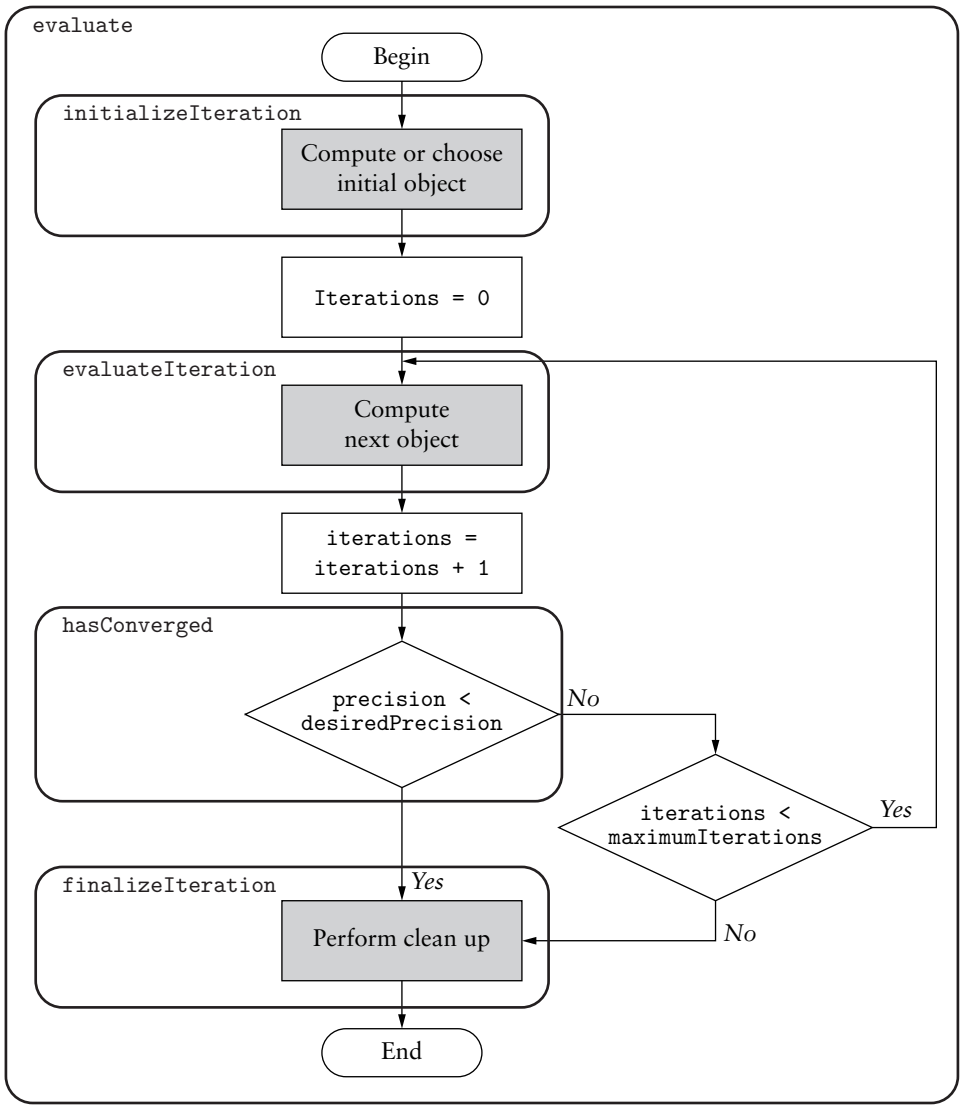


**FIG. 4.4** Methods for successive approximations

methods defined for an iterative process. The precision of the attained result and the number of iterations are printed on the transcript window.

**Code Example 4.2**

```
| iterativeProcess result precision |
iterativeProcess := <a subclass of DhbIterativeProcess> new.
iterativeProcess desiredPrecision: 1.0e-6; maximumIterations: 25.
```

```
result := iterativeProcess evaluate.
iterativeProcess hasConverged
ifTrue: [ Transcript nextPutAll: 'Result obtained after '.
iterativeProcess iteration printOn: Transcript.
Transcript nextPutAll: 'iterations. Attained precision is '.
iterativeProcess precision printOn: Transcript.
]
ifFalse:[ Transcript nextPutAll: 'Process did not converge'.].
Transcript cr.
```

Listing 4.1 shows the Smalltalk implementation of the iterative process.

In the Smalltalk implementation, the class `IterativeProcess` has one instance variable in addition to the ones described in the preceding section. This variable, called `result`, is used to keep the solution object of the process. The method `result` allows direct access to it. Thus, all subclasses can use this instance variable as a placeholder to store any type of result. As a convenience, the method `evaluate` also returns the instance variable `result`.

Default values for the desired precision and the maximum number of iterations are kept in class methods for easy editing. The method `initialize` loads these default values for each newly created instance. The default precision is set to the machine precision discussed in Section 1.3.2.

The methods used to modify the desired precision (`desiredPrecision:`) and the maximum number of iterations (`maximumIterations:`) check the value to prevent illegal definitions, which could prevent the algorithm from terminating.

Since there is no explicit declaration of abstract class and abstract methods in Smalltalk,[2] the three methods `initializeIterations`, `evaluateIteration` and `finalizeIterations` are implemented with a reasonable default behavior. The methods `initializeIterations` and `finalizeIterations` do nothing. The method `evaluateIteration` calls the method `subclassResponsibility`, which raises an exception when called. Using this technique is Smalltalk's way of creating an abstract method.

**Listing 4.1 Smalltalk implementation of an iterative process**

| | |
|---|---|
| *Class* | `DhbIterativeProcess` |
| *Subclass of* | `Object` |
| *Instance variable names:* | `precision desiredPrecision maximumIterations result iterations` |

2. An *abstract* class is a class containing at least an abstract method; an abstract method contains the single conventional statement: `self subclassResponsibility`.

*Class Methods*

**defaultMaximumIterations**

```
^50
```

**defaultPrecision**

```
^DhbFloatingPointMachine new defaultNumericalPrecision
```

**new**

```
^super new initialize
```

*Instance Methods*

**desiredPrecision:**

```
aNumber

aNumber > 0
    ifFalse: [ ^self error: 'Illegal precision: ',
                                            aNumber printString].
desiredPrecision := aNumber.
```

**evaluate**

```
iterations := 0.
self initializeIterations.
[iterations := iterations + 1.
precision := self evaluateIteration.
self hasConverged or: [iterations >= maximumIterations]]
        whileFalse: [].
self finalizeIterations.
^self result
```

**evaluateIteration**

```
^self subclassResponsibility
```

**finalizeIterations**

**hasConverged**

```
^precision <= desiredPrecision
```

**initialize**

```
desiredPrecision := self class defaultPrecision.
maximumIterations := self class defaultMaximumIterations.
^self
```

**initializeIterations**

**iterations**

```
^iterations
```

**maximumIterations:**

```
anInteger
```

```
( anInteger isInteger and: [ anInteger > 1])
    ifFalse: [ ^self error: 'Invalid maximum number of iteration: '
                                        , anInteger printString].
maximumIterations := anInteger.
```

**precision**

```
^precision
```

**precisionOf:**

```
aNumber1
```

**relativeTo:**

```
aNumber2
```

```
^aNumber2 > DhbFloatingPointMachine new defaultNumericalPrecision
    ifTrue: [ aNumber1 / aNumber2]
    ifFalse:[ aNumber1]
```

**result**

```
^result
```

---

**Note:** The method `precisionOf:relativeTo:` implements the computation of the relative precision. This topic is discussed in Section 4.2.1.

### 4.1.2 Iterative Process—JAVA Implementation

Even though we are dealing for the moment with an abstract class, we are able to present a scenario of use illustrating the public interface to the class. Code Example 4.3 shows what a basic utilization of an iterative process object would look like.

**Code Example 4.3**

```
<a subclass of IterativeProcess> iterativeProcess
= new <a subclass of IterativeProcess>();
iterativeProcess.evaluate();
if ( iterativeProcess.hasConverged) {
```

```
    :
    <retrieve the result>;
} else {
    :
    <special case processing>
}
```

The first statement creates an object to handle the iterative process. The second one performs the process. The final statement checks for convergence.

To give the user a possibility to have more control, one can extend the public interface of the object to allow defining the parameters of the iterative process: the desired precision and the maximum number of iterations. In addition, the user may want to know the precision of the attained result and the number of iterations needed to obtain the result. Code Example 4.4 shows an example of use for all public methods defined for an iterative process. The precision of the attained result and the number of iterations are printed on the console output.

**Code Example 4.4**

```
| iterativeProcess result |
<a subclass of IterativeProcess> iterativeProcess =
  new <a subclass of IterativeProcess>();
iterativeProcess. setMaximumIterations (25);
iterativeProcess. setDesiredPrecision (1.0e-6);
iterativeProcess.evaluate();
if ( iterativeProcess.hasConverged) {
System.out.println( "Result obtained after
"+iterativeProcess.getIterations()+" iterations. Attained
precision is "+iterativeProcess.getPrecision());
} else {
System.out.println( "Process did not converge");
}
```

Listing 4.2 shows the Java implementation of the iterative process.

The instance variables are kept private since no subclass should be able to modify them. Otherwise, the efficacy of the algorithm could be disturbed.

In the case of a general-purpose algorithm, the nature of the result is not known. It would be possible to declare a variable of type `Object` to hold a solution object of any type. However, this would force each subclass and each client of the subclass to use an explicit cast to obtain the result into the proper class. In general, casts ought to be avoided because they beat the purpose of using strong typing, which is checking code consistency at compile time. Thus, unlike the Smalltalk implementation, no instance variable can be used to hold the result.

For the same reason, the method `evaluate` cannot return the result. It is thus declared as void. Only the subclass knows the exact type of the result. A method

`getResult` must be implemented by each subclass to allow the client class to retrieve the result.

The method `evaluateIteration` is declared as an abstract method to make sure that any subclass implements it. The methods `initializeIterations` and `finalizeIterations` are supplied with a default behavior, which is to do nothing.

Default values for the desired precision and the maximum number of iterations are assigned statically. Any subclass needing to change these values must use the corresponding setting methods in its constructor method.

Other than these points, the Java implementation is identical to the Smalltalk one.

**Listing 4.2 Java implementation of an iterative process**

```
package DhbIterations;

import DhbFunctionEvaluation.DhbMath;

// An iterative process is a general structure managing iterations.

// @author Didier H. Besset

public abstract class IterativeProcess
{

    // Number of iterations performed.

    private int iterations;

    // Maximum allowed number of iterations.

    private int maximumIterations = 50;

    // Desired precision.

    private double desiredPrecision = DhbMath.defaultNumericalPrecision();

    // Achieved precision.

    private double precision;

// Generic constructor.

public IterativeProcess() {
}
```

```
// Performs the iterative process.
// Note: this method does not return anything because Java does not
// allow mixing double, int, or objects

public void evaluate()
{
    iterations = 0;
    initializeIterations();
    while ( iterations++ < maximumIterations )
    {
        precision = evaluateIteration();
        if ( hasConverged() )
            break;
    }
    finalizeIterations();
}

// Evaluate the result of the current interation.
// @return the estimated precision of the result.

abstract public double evaluateIteration();

// Perform eventual clean-up operations
// (mustbe implemented by subclass when needed).

public void finalizeIterations ( )
{
}

// Returns the desired precision.

public double getDesiredPrecision( )
{
    return desiredPrecision;
}

// Returns the number of iterations performed.

public int getIterations()
{
    return iterations;
}

// Returns the maximum allowed number of iterations.

public int getMaximumIterations( )
```

```
{
    return maximumIterations;
}

// Returns the attained precision.

public double getPrecision()
{
    return precision;
}

// Check to see if the result has been attained.
// @return boolean

public boolean hasConverged()
{
    return precision < desiredPrecision;
}

// Initializes internal parameters to start the iterative process.

public void initializeIterations()
{
}

// @return double
// @param epsilon double
// @param x double

public double relativePrecision( double epsilon, double x)
{
    return x > DhbMath.defaultNumericalPrecision()
                                            ? epsilon / x: epsilon;
}

// Defines the desired precision.

public void setDesiredPrecision( double prec )
                                    throws IllegalArgumentException
{
    if ( prec <= 0 )
        throw new IllegalArgumentException
                                ( "Non-positive precision: "+prec);
    desiredPrecision = prec;
}
```

```
    // Defines the maximum allowed number of iterations.

    public void setMaximumIterations( int maxIter)
                                        throws IllegalArgumentException
    {
        if ( maxIter < 1 )
            throw new IllegalArgumentException
                            ( "Non-positive maximum iteration: "+maxIter);
        maximumIterations = maxIter;
    }
    }
```

**Note:** The method `relativePrecision` implements the computation of the relative precision. This is discussed in Section 4.2.2.

## 4.2 Evaluation with Relative Precision

So far we have made no assumption about the nature of the solution sought by an iterative process. In this section I want to discuss the case when the solution is a numerical value.

As discussed in Section 1.3.2, a floating-point number is a representation with constant relative precision. It is thus meaningless to use absolute precision to determine the convergence of an algorithm. The precision of an algorithm resulting in a numerical value ought to be determined relatively.

One way to do it is to have the method `evaluateIteration` returning a relative precision instead of an absolute number. Relative precision, however, can only be evaluated if the final result is different from zero. If the result is zero, the only possibility is to check for absolute precision. Of course, in practice one does not check for equality with zero. The computation of a relative precision is carried only if the absolute value of the result is larger than the desired precision.

The reasoning behind the computation of the relative error is quite general. Thus, a general-purpose class `FunctionalIterator` has been created to implement a method computing the relative precision from an absolute precision and a numerical result. In addition, since all subclasses of `FunctionalIterator` use a function, a general method to handle the definition of that function is also supplied.

### 4.2.1 Relative Precision—Smalltalk Implementation

In this case, the public interface is extended with a creation method taking as argument the function on which the process operates. Code Example 4.1 then becomes what appears in Code Example 4.5.

**Code Example 4.5**

```
| iterativeProcess result |
iterativeProcess := <a subclass of DhbFunctionalIterator> function:
( DhbPolynomial coefficients: #(1 2 3).
result := iterativeProcess evaluate.
iterativeProcess hasConverged
ifFalse:[ <special case processing>].
```

In this example, the function on which the process will operate is the polynomial $3x^2 + 2x + 1$ (see Section 2.2).

Listing 4.3 shows the implementation of the abstract class `DhbFunctionalIterator` in Smalltalk.

This class has one instance variable `functionBlock` to store the function. A single class method allows creating a new instance while defining the function.

As we have seen in Section 2.1.1, a function can be any object responding to the message `value:`. This allows supplying any block of Smalltalk code as argument to the constructor method. However, the user can also supply a class implementing the computation of the function with a method with selector `value:`. For example, an instance of the class `DhbPolynomial` discussed in Section 2.2.3 can be used.

The instance method `setFunction:` is used to set the instance variable `functionBlock`. To prevent a client class from sending the wrong object, the method first checks whether the supplied object responds to the message `value:`. This is one way of ensuring that the arguments passed to a method conform to the expected protocol. This approach is only shown as an example, however, it is not recommend in practice. The responsibility of supplying the correct arguments to a Smalltalk method is usually the responsibility of the client class.

The method `initializeIterations` first checks whether a function block has been defined. Then it calls the method `computeInitialValues`. This method is a hook method, which a subclass must implement to compute the value of the result at the beginning of the iterative process.

The computation of relative precision is implemented at two levels. One general method, `precisionOf:relativeTo:`, implemented by the superclass allows the computation of the relative precision relative to any value. Any iterative process can use this method. The method `relativePrecision` implements the computation of the precision relative to the current result.

---

**Listing 4.3** **Smalltalk implementation of the class** `DhbFunctionalIterator`

| | |
|---|---|
| *Class* | `DhbFunctionalIterator` |
| *Subclass of* | `DhbIterativeProcess` |
| *Instance variable names:* | `functionBlock relativePrecision` |

*Class Methods*

**function:**

```
aBlock

^self new setFunction: aBlock; yourself
```

*Instance Methods*

**initializeIterations**

```
functionBlock isNil ifTrue: [self error: 'No function supplied'].
self computeInitialValues
```

**relativePrecision:**

```
aNumber

^self precisionOf: aNumber relativeTo: result abs
```

**setFunction:**

```
aBlock

( aBlock respondsTo: #value:)
    ifFalse:[ self error: 'Function block must implement the
                                                    method value:'].
functionBlock := aBlock.
```

---

## 4.2.2 Relative Precision—Java Implementation

In this case, the public interface is extended with a creation method taking as argument the function on which the process operates. Code Example 4.3 then becomes the depiction in Code Example 4.6.

**Code Example 4.6**

```
double[]coefficients = 1, 2, 3;
<a subclass of FunctionalIterator> iterativeProcess =
  new <a subclass of FunctionalIterator>(new Polynomial(coefficients));
iterativeProcess.evaluate();
if ( iterativeProcess.hasConverged) {
double result = iterativeProcess.getResult();
} else {
    .
    .   <special case processing>
    .
}
```

In this example, the function on which the process will operate is the polynomial $3x^2 + 2x + 1$ (see Section 2.2). In the case of a numerical result, the type of the result is known; thus, we can retrieve the result explicitly.

Listing 4.4 shows the implementation of the abstract class `FunctionalIterator` in Java.

In Java the abstract class has two instances variables: one to hold the function, as in Smalltalk, and one to hold the result now that the type of the result is defined (see the remark in Section 4.1.2). An access method for the result is implemented.

The computation of relative precision is implemented at two levels. One general method, `relativePrecision` with two arguments, implemented by the superclass allows the computation of the relative precision relative to any value. Any iterative process can use this method. The method `relativePrecision` with one argument implements the computation of the precision relative to the current result.

Other than these points, the implementation is similar to that of Smalltalk.

**Listing 4.4 Java implementation of the class**

`FunctionalIterator`

```
package DhbIterations;

import DhbInterfaces.OneVariableFunction;

// Iterative process based on a one-variable function,
// having a single numerical result.

// @author Didier H. Besset

public abstract class FunctionalIterator extends IterativeProcess
{

    // Best approximation of the zero.

    protected double result = Double.NaN;

    // Function for which the zero will be found.

    protected OneVariableFunction f;

// Generic constructor.
// @param func OneVariableFunction
// @param start double

public FunctionalIterator(OneVariableFunction func)
{
```

```
        setFunction( func);
    }

    // Returns the result (assuming convergence has been attained).

    public double getResult( )
    {
        return result;
    }

    // @return double
    // @param epsilon double

    public double relativePrecision( double epsilon)
    {
        return relativePrecision( epsilon, Math.abs( result));
    }

    // @param func DhbInterfaces.OneVariableFunction

    public void setFunction( OneVariableFunction func)
    {
        f = func;
    }

    // @param x double

    public void setInitialValue( double x)
    {
        result = x;
    }
    }
```

## 4.3 Examples

As we have dealt with abstract classes, this chapter did not give concrete examples of use. By consulting the rest of this book, the reader will find numerous examples of subclasses of the two classes described in this chapter. Table 4.1 lists the sections in which each algorithm using the iterative process framework is discussed.

**TABLE 4.1** Algorithms using iterative processes

| Algorithm or class of algorithm | Superclass | Chapter or section |
|---|---|---|
| Zero finding | Function iterator | Chapter 5 |
| Integration | Function iterator | Chapter 6 |
| Infinite series and continued fractions | Function iterator | Chapter 7 |
| Matrix eigenvalues | Iterative process | Section ??? |
| Nonlinear least-square fit | Iterative process | Section ??? |
| Maximum-likelihood fit | Iterative process | Section ??? |
| Function minimization | Function iterator | Chapter ??? |
| Cluster analysis | Iterative process | Section ??? |

CHAPTER 5

# Finding the Zero of a Function

*Le zéro, collier du néant.*[1]
—Jean Cocteau

The *zeroes* of a function are the values of the function's variable for which the value of the function is zero. Mathematically, given the function $f(x)$, $z$ is a zero when $f(z) = 0$. This kind of problem can be extended to the general problem of computing the value of the inverse function—that is, finding a value $x$ such that $f(x) = c$, where $c$ is a given number. The inverse function is noted as $f^{-1}(x)$. Thus, one wants to find the value of $f^{-1}(c)$ for any $c$. The problem can be transformed into the problem of finding the zero of the function $\tilde{f}(x) = f(x) - c$.

The problem of finding the values at which a function takes a maximum or minimum value is called *searching for the extremes* of a function. This problem can be transformed into a zero-finding problem if the derivative of the function can be easily computed. The extremes are the zeroes of the function's derivative.

Figure 5.1 shows the class diagram of the classes discussed in this chapter.

## 5.1 Introduction

Let us begin with a concrete example. Often an experimental result is obtained by measuring the same quantity several times. In scientific publications, such a result is published with two numbers: the average and the standard deviation of the measurements. This is true for medical publication as well. As we have already discussed in Section 2.3.1, obstetricians prefer to think in terms of risk and prefer to use centiles instead of average and standard deviation. Assuming that the measurements were distributed according to a normal distribution (see Section ???), the 90th centile is the solution to equation 5.1:

$$\text{erf}(x) = 0.9 \tag{5.1}$$

1. The zero, a necklace for emptiness.

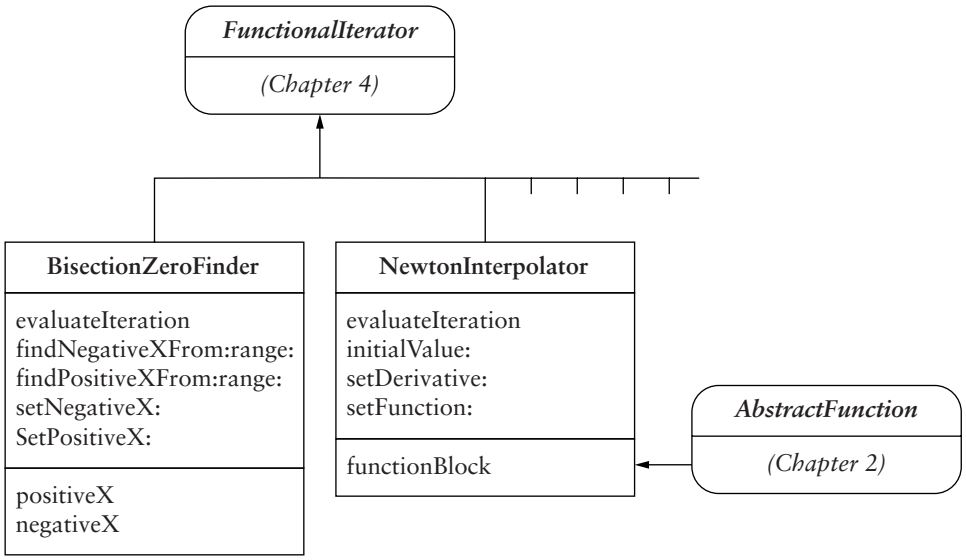


**FIG. 5.1** Class diagram for zero finding classes

That is, we need to find the zero of the function $f(x) = \text{erf}(x) - 0.9$. The answer is $x = 1.28$ with a precision of two decimals. Thus, if $\mu$ and $\sigma$ are, respectively, the average and standard deviation of a published measurement, the 90th centile is given by $\mu + 1.28 \cdot \sigma$. Using equation 2.19, the 10th centile is given by $\mu - 1.28 \cdot \sigma$.

## 5.2 Finding the Zeroes of a Function– Bisection Method

Let assume that one knows two values of $x$ for which the function takes values of opposite sign. Let us call $x_{\text{pos}}$ the value such that $f(x_{\text{pos}}) > 0$ and $x_{\text{neg}}$ the value such that $f(x_{\text{neg}}) < 0$. If the function is continuous between $x_{\text{pos}}$ and $x_{\text{neg}}$, at least one zero of the function exists in the interval $[x_{\text{pos}}, x_{\text{neg}}]$. This case is illustrated in Figure 5.2. If the function $f$ is not continuous over the interval where the sign of the function changes, then the presence of a zero cannot be guaranteed.[2] The continuity requirement is essential for the application of the bisection algorithm.

The values $x_{\text{pos}}$ and $x_{\text{neg}}$ are the initial values of the bisection algorithm. The algorithm goes as follows:

2. The inverse function is such an example. It changes sign over zero but has no zeroes for any finite $x$.

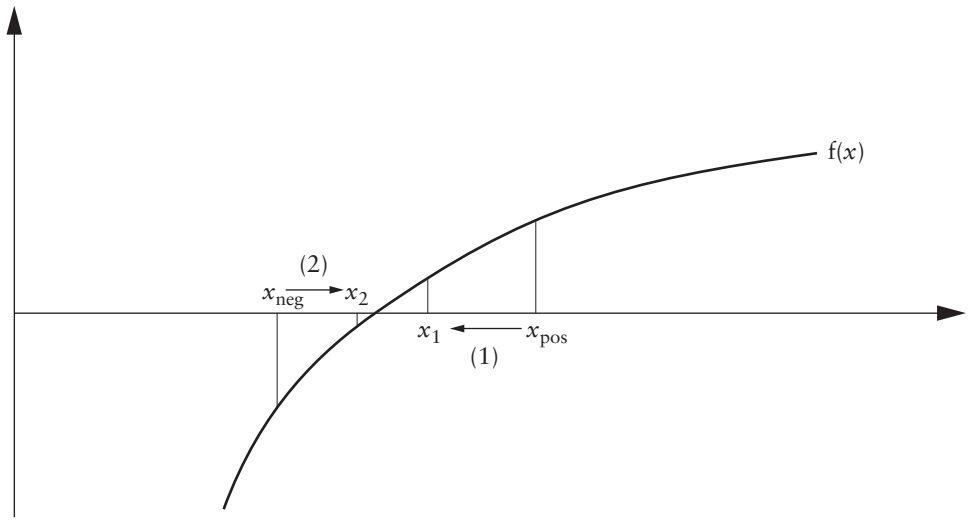


**FIG. 5.2** The bisection algorithm

1. Compute $x = \frac{x_{pos} - x_{neg}}{2}$.
2. If $f(x) > 0$, set $x_{pos} = x$ and go to step 4.
3. Otherwise, set $x_{neg} = x$.
4. If $|x_{pos}, x_{neg}| > \epsilon$ go back to step 1, $\epsilon$ is the desired precision of the solution.

The first couple of steps of the bisection algorithm are represented geometrically in Figure 5.2. Given the two initial values, $x_{pos}$ and $x_{neg}$, the first iteration of the algorithm replaces $x_{pos}$ with $x_1$. The next step replaces $x_{neg}$ with $x_2$.

For a given pair of initial values, $x_{pos}$ and $x_{neg}$, the number of iterations required to attain a precision $\epsilon$ is given by equation 5.2:

$$n = \lceil \log_2 \frac{|x_{pos}, x_{neg}|}{\epsilon} \rceil. \tag{5.2}$$

For example, if the distance between the two initial values is 1, the number of iterations required to attain a precision of $10^{-8}$ is 30. It shows that the bisection algorithm is rather slow.

Knowledge of the initial values, $x_{pos}$ and $x_{neg}$, is essential for starting the algorithm. Methods to define them must be supplied. Two convenience methods are supplied to sample the function randomly over a given range to find each initial value. (The random number generator is discussed in Section ???.)

The bisection algorithm is a concrete implementation of an iterative process. In this case, the method `evaluateIteration` of Figure 4.4 implements steps 2, 3,

and 4. The precision at each iteration is $|x_{pos}, x_{neg}|$ since the zero of the function is always inside the interval defined by $x_{pos}$ and $x_{neg}$.

### 5.2.1 Bisection Algorithm—General Implementation

The class of the object implementing the bisection algorithm is a subclass of the abstract class `FunctionalIterator`. The class `BisectionZeroFinder` needs the following additional instance variables.

`positiveX` $x_{pos}$

`negativeX` $x_{neg}$

The bisection algorithm proper is implemented only within the method `evaluateIteration`. Other necessary methods have already been implemented in the iterative process class.

### 5.2.2 Bisection Algorithm—Smalltalk Implementation

Finding the zero of a function is performed by creating an instance of the class `DhbBisectionZeroFinder` and giving the function as the argument of the creation method, as explained in Section 4.2.1. For example, the code in Code Example 5.1 finds the solution of equation 5.1.

**Code Example 5.1**

```
| zeroFinder result |
zeroFinder:= DhbBisectionZeroFinder function:
[ :x | x errorFunction - 0.9].
zeroFinder setPositiveX: 10; setNegativeX: 0.
result := zeroFinder evaluate.
zeroFinder hasConverged
  ifFalse:[ <special case processing>].
```

The second line creates the object responsible to find the zero. The third line defines the initial values, $x_{pos}$ and $x_{neg}$. The fourth line performs the algorithm and stores the result if the algorithm has converged. The last two lines check for convergence and take corrective action if the algorithm did not converge.

Listing 5.1 shows the implementation of the bisection zero finding algorithm in Smalltalk.

The class `DhbBisectionZeroFinder` is a subclass of the class `DhbFunctionalIterator`. As one can see, only a few methods need to be implemented. Most of them pertain to the definition of the initial interval. In particular, convenience methods are supplied to find a positive and negative function value over a given interval.

The methods defining the initial values, $x_{pos}$ and $x_{neg}$, are `setPositiveX:` and `setNegativeX:`, respectively. An error is generated in each method if the func-

tion's value does not have the proper sign. The convenience methods to find random starting values are, respectively, `findPositiveXFrom:range:` and `findNegativeXFrom:range:`. The method `computeInitialValues` does not compute the initial values. Instead, it makes sure that $x_{\text{pos}}$ and $x_{\text{neg}}$ have been properly defined.

---

**Listing 5.1 Smalltalk implementation of the bisection algorithm**

| | |
|---|---|
| *Class* | `DhbBisectionZeroFinder` |
| *Subclass of* | `DhbFunctionalIterator` |
| *Instance variable names:* | `positiveX negativeX` |

***Instance Methods***

**computeInitialValues**

```
positiveX isNil
    ifTrue: [ self error: 'No positive value supplied'].
negativeX isNil
    ifTrue: [ self error: 'No negative value supplied'].
```

**evaluateIteration**

```
result := ( positiveX + negativeX) * 0.5.
( functionBlock value: result) > 0
    ifTrue: [ positiveX := result]
    ifFalse:[ negativeX := result].
^self relativePrecision: ( positiveX - negativeX) abs
```

**findNegativeXFrom:** `aNumber1` **range:** `aNumber2`

```
| n |
n := 0.
[ negativeX := Number random * aNumber2 + aNumber1.
  ( functionBlock value: negativeX) < 0
    ] whileFalse: [ n := n + 0.1.
                    n > maximumIterations
                        ifTrue: [ self error: 'Unable to find a
                                        negative function value'].
                  ].
```

**findPositiveXFrom:** `aNumber1` **range:** `aNumber2`

```
| n |
n := 0.
[ positiveX := Number random * aNumber2 + aNumber1.
  ( functionBlock value: positiveX) > 0
```

```
    ] whileFalse: [ n := n + 1.
                    n > maximumIterations
                        ifTrue: [ self error: 'Unable to find a
                                        positive function value'].
                  ].
```

**setNegativeX:** aNumber

```
( functionBlock value: aNumber) < 0
    ifFalse:[ self error: 'Function is not negative at x = ',
                                            aNumber printString].
negativeX := aNumber.
```

**setPositiveX:** aNumber

```
( functionBlock value: aNumber) > 0
    ifFalse:[ self error: 'Function is not positive at x = ',
                                            aNumber printString].
positiveX := aNumber.
```

---

### 5.2.3 Bisection Algorithm—Java Implementation

Finding the zero of a function is performed by creating an instance of the class `BisectionZeroFinder` and giving the function as the argument of the constructor method, as explained in Section 4.2.2. For example, the code in Code Example 5.2 finds the solution of equation 5.1.

**Code Example 5.2**

```
BisectionZeroFinder zeroFinder = new BisectionZeroFinder( new
       OneVariableFunction() { public double value( double x)
       { return NormalDistribution.errorFunction(x)-0.9;}});
try { zeroFinder.setNegativeX( 0);
      zeroFinder. setPositiveX ( 5);
    } catch( IllegalArgumentException e) { return};
zeroFinder.evaluate();
double result = zeroFinder.getResult();
```

The first line creates the object responsible to find the zero. It uses an inner class to define the logarithm function. The second line defines the initial values, $x_{\text{pos}}$ and $x_{\text{neg}}$, within an exception-catching construct (`try...catch`). The line before the last one performs the algorithm. The last line retrieves the result assuming the algorithm has converged.

Listing 5.2 shows the implementation of the bisection zero-finding algorithm in Java.

The class `BisectionZeroFinder` is a subclass of the class `FunctionalIterator`. As one can see, only a few methods need to be implemented. Most of them pertain to the definition of the initial interval.

The methods defining the initial values, $x_{\text{pos}}$ and $x_{\text{neg}}$, are, respectively, `setPositiveX` and `setNegativeX`. Both of them throw an illegal argument exception if the function's value does not have the proper sign. The convenience methods to find random starting values are, respectively, `findPositiveX` and `findNegativeX`.

**Listing 5.2 Java implementation of a generic derivative evaluation**

```
package DhbIterations;

import DhbInterfaces.OneVariableFunction;

// Zero finding by bisection.

// @author Didier H. Besset

public class BisectionZeroFinder extends FunctionalIterator
{

    // Value at which the function's value is negative.

    private double xNeg;

    // Value at which the function's value is positive.

    private double xPos;

// @param func DhbInterfaces.OneVariableFunction

public BisectionZeroFinder(DhbInterfaces.OneVariableFunction func) {
    super(func);
}

// @param func DhbInterfaces.OneVariableFunction
// @param x1 location at which the function yields a negative
value
// @param x2 location at which the function yields a positive
value

public BisectionZeroFinder( OneVariableFunction func, double x1,
double x2) throws IllegalArgumentException
{
```

```
    this(func);
    setNegativeX( x1);
    setPositiveX( x2);
}

// @return double

public double evaluateIteration()
{
    result = ( xPos + xNeg) * 0.5;
    if ( f.value(result) > 0 )
        xPos = result;
    else
        xNeg = result;
    return relativePrecision( Math.abs( xPos - xNeg));
}

// @param x double
// @exception java.lang.IllegalArgumentException
//              if the function's value is not negative

public void setNegativeX( double x) throws IllegalArgumentException
{
    if ( f.value( x) > 0 )
        throw new IllegalArgumentException( "f("+x+
                                ") is positive instead of negative");
    xNeg = x;
}

// (c) Copyrights Didier BESSET, 1999, all rights reserved.
// @param x double
// @exception java.lang.IllegalArgumentException
//              if the function's value is not positive

public void setPositiveX( double x) throws IllegalArgumentException
{
    if ( f.value( x) < 0 )
        throw new IllegalArgumentException( "f("+x+
                                ") is negative instead of positive");
    xPos = x;
}
}
```

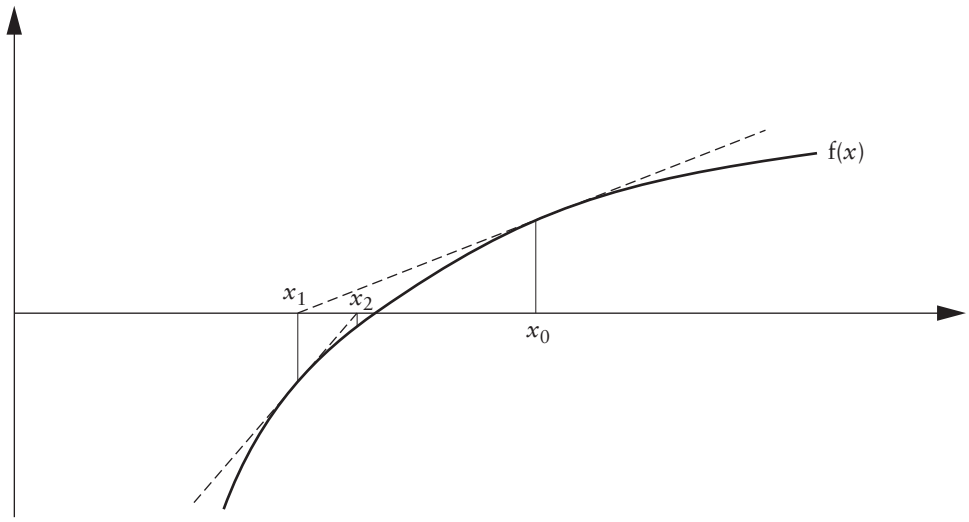


**FIG. 5.3** Geometrical representation of Newton's zero finding algorithm

## 5.3 Finding the Zero of a Function—Newton's Method

Isaac Newton designed an algorithm working by successive approximations [Bass]. Given a value $x_0$ chosen in the vicinity of the desired zero, the following series:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, \tag{5.3}$$

where $f'(x)$ is the first derivative of $f(x)$, converges toward a zero of the function. This algorithm is sometimes called *Newton-Ralphson* [Press *et al.*].

Figure 5.3 shows the geometrical interpretation of the series. $f'(x)$ is the slope of the tangent to the curve of the function $f(x)$ at the point $x_n$. The equation of this tangent is thus given by:

$$y = (x - x_n) \cdot f'(x_n) + f(x_n). \tag{5.4}$$

One can then see that $x_{n+1}$ is the point where the tangent to the curve at the point $x_n$ crosses the $x$-axis. The algorithm can be started at any point where the function's derivative is nonzero.

The technique used in Newton's algorithm is a general one often used in approximations. The function is replaced by a linear approximation[3]—that is, a straight line

3. Mathematically, this corresponds to an estimate of the function using the first two terms of its Taylor series.

going through the point defined by the preceding value and its function's value. The slope of the straight line is given by the first derivative of the function. The procedure is repeated until the variation between the new value and the preceding one is sufficiently small. We shall see other examples of this technique in the remainder of this book (see Sections ???, ???, and ???).

From equation 5.3, one can see that the series may not converge if $f'(x)$ becomes zero. If the derivative of the function is zero in the vicinity of the zero, the bisection algorithm gives better results. Otherwise, Newton's algorithm is highly efficient. It usually requires 5 to 10 times less iteration than the bisection algorithm, which largely compensates for the additional time spent in computing the derivative.

The class implementing Newton's algorithm belongs to a subclass of the functional iterator described in Section 4.2. An additional instance variable is needed to store the function's derivative.

### 5.3.1 Newton's Method—Smalltalk Implementation

Listing 5.3 shows the complete implementation in Smalltalk. The class `DhbNewtonZeroFinder` is a subclass of the class `DhbFunctionalIterator` described in Section 4.2.1. For example, Code Example 5.3 finds the solution of equation 5.1. The second line creates the object responsible to find the zero supplying the function and the derivative.[4] The third line defines the starting value. The fourth line performs the algorithm and stores the result if the algorithm has converged. The last two lines check for convergence and take corrective action if the algorithm did not converge.

**Code Example 5.3**

```
| zeroFinder result | zeroFinder:= DhbNewtonZeroFinder
            function: [ :x | x errorFunction - 0.9]
            derivative: [ :x | DhbErfApproximation new normal: x].
zeroFinder initialValue: 1.
result := zeroFinder evaluate.
zeroFinder hasConverged
    ifFalse:[ <special case processing>].
```

The method `computeInitialValues` is somewhat complex. First, it checks whether the user supplied an initial value. If not, it is assigned to zero. Then it checks whether the user supplied a derivative. If not, a default derivative function is supplied as a block closure by the method `defaultDerivativeBlock`. The supplied block closure implements the formula of equation 2.4. If a derivative is supplied, it is compared to the result of the derivative supplied by default, which may save a lot of trouble if the user made an error in coding the derivative. Not supplying a derivative has some negative effect on the speed and limits the precision of the final result. The method `initializeIterations` also checks whether the derivative is nearly zero

4. As we have seen in Section 2.3, the normal distribution is the derivative of the error function.

for the initial value. If that is the case, the initial value is changed with a random walk algorithm. If no value can be found such that the derivative is nonzero, an error is generated.

If the function is changed, the supplied derivative must be suppressed. Thus, the method `setFunction:` must also force a redefinition of the derivative. A method allows defining the initial value. A creation method defining the function and derivative is also supplied for convenience.

As for the bisection, the algorithm itself is coded within the method `evaluateIteration`. Other methods needed by the algorithm have already been implemented in the superclasses.

---

**Listing 5.3 Smalltalk implementation of Newton's zero-finding method**

| | |
|---|---|
| *Class* | `DhbNewtonZeroFinder` |
| *Subclass of* | `DhbFunctionalIterator` |
| *Instance variable names:* | `derivativeBlock` |

***Class Methods***

**function:** `aBlock1` **derivative:** `aBlock2`

```
^(self new) setFunction: aBlock1; setDerivative: aBlock2;
                                                        yourself
```

Instance methods

**computeInitialValues**

```
| n |
result isNil
    ifTrue: [ result := 0].
derivativeBlock isNil
    ifTrue: [ derivativeBlock := self defaultDerivativeBlock].
n := 0.
[ (derivativeBlock value: result) equalsTo: 0]
    whileTrue: [ n := n + 1.
                 n > maximumIterations
                    ifTrue: [ self error: 'Function''s derivative
                                    seems to be zero everywhere'].
                 result := Number random + result].
```

**defaultDerivativeBlock**

```
^[ :x | 5000 * ( ( functionBlock value: (x + 0.0001)) -
                            ( functionBlock value: (x - 0.0001)))]
```

**evaluateIteration**

```
| delta |
delta := ( functionBlock value: result) /
                                ( derivativeBlock value: result).
result := result - delta.
^self relativePrecision: delta abs
```

**initialValue:** aNumber

```
result := aNumber.
```

**setDerivative:** aBlock

```
| x |
( aBlock respondsTo: #value:)
    ifFalse:[ self error: 'Derivative block must implement the
                                                    method value:'].
x := result ifNil: [ Number random] ifNot: [ :base | base +
                                                     Number random].
( ( aBlock value: x) relativelyEqualsTo:
                    (self defaultDerivativeBlock value: x) upTo: 0.0001)
    ifFalse:[ self error: 'Supplied derivative is not correct'].
derivativeBlock := aBlock.
```

**setFunction:** aBlock

```
super setFunction: aBlock.
derivativeBlock := nil.
```

---

## 5.3.2 Newton's Method–Java Implementation

The class is a subclass of the class FunctionalIterator described in Section 4.3. For example Code Example 5.4 finds the solution of equation 5.1.

**Code Example 5.4**

```
NewtonZeroFinder zeroFinder = new NewtonZeroFinder(
    new OneVariableFunction() { public double value( double x)
    { return NormalDistribution.errorFunction(x) - 0.9;}});
zeroFinder.setDerivative(new OneVariableFunction()
    { public double value( double x)
    { return NormalDistribution.normal(x);}});
zeroFinder. setStartingValue( 1.0); zeroFinder.evaluate();
double result = zeroFinder.getResult();
```

The first line creates the object responsible to find the zero. It uses an inner class to define the error function. The second line defines the derivative also defined as

an inner class. The third line defines the starting value. The line before the last one performs the algorithm. The last line retrieves the result assuming the algorithm has converged.

Listing 5.4 shows the complete implementation in Java.

The strong typing of Java saves us the need for checking supplied parameters. The supplied function and derivative must implement the `OneVariableFunction` interface (see Section 2.1.2). If the derivative function is not supplied, an object constructed with the class `FunctionDerivative` (see Section 2.1.2) is used instead.

Since the superclass of `NewtonZeroFinder` is an abstract class, two constructor methods are supplied. One defines the function and the initial value, the second defines the derivative also.

Other than these points the code is similar to that of the Smalltalk implementation.

**Listing 5.4 Java implementation of Newton's zero-finding method**

```
package DhbIterations;

import DhbFunctionEvaluation.DhbMath;
import DhbFunctionEvaluation.FunctionDerivative;
import DhbInterfaces.OneVariableFunction;

// Finds the zeroes of a function using Newton approximation.
// Note: the zero of a function if the value at which the function's
// value is zero.

// @author Didier H. Besset

public class NewtonZeroFinder extends FunctionalIterator
{

    // Derivative of the function for which the zero will be found.

    private OneVariableFunction df;

// Constructor method.
// @param func the function for which the zero will be found.
// @param start the initial value for the search.

public NewtonZeroFinder( OneVariableFunction func, double start)
{
    super( func);
    setStartingValue( start);
}
```

```
// Constructor method.
// @param func the function for which the zero will be found.
// @param dFunc derivative of func.
// @param start the initial value for the search.

public NewtonZeroFinder( OneVariableFunction func,
                         OneVariableFunction dFunc, double start)
                                 throws IllegalArgumentException
{
    this( func, start);
    setDerivative( dFunc);
}

// Evaluate the result of the current interation.
// @return the estimated precision of the result.

public double evaluateIteration()
{
    double delta = f.value( result) / df.value( result);
    result -= delta;
    return relativePrecision( Math.abs( delta));
}

// Initializes internal parameters to start the iterative process.
// Assigns default derivative if necessary.

public void initializeIterations()
{
    if ( df == null)
        df = new FunctionDerivative( f);
    if ( Double.isNaN( result) )
        result = 0;
    int n = 0;
    while ( DhbMath.equal( df.value( result), 0) )
    {
        if ( ++n > getMaximumIterations() )
            break;
        result += Math.random();
    }
}

// (c) Copyrights Didier BESSET, 1999, all rights reserved.
// @param dFunc DhbInterfaces.OneVariableFunction
// @exception java.lang.IllegalArgumentException
//                  if the derivative is not accurate.
```

```
public void setDerivative( OneVariableFunction dFunc)
throws IllegalArgumentException
{
    df = new FunctionDerivative( f);
    if ( !DhbMath.equal( df.value( result), dFunc.value( result), 0.001)
)
        throw new IllegalArgumentException
                        ( "Supplied derivative function is inaccurate");
    df = dFunc;
}

// (c) Copyrights Didier BESSET, 1999, all rights reserved.

public void setFunction( OneVariableFunction func)
{
    super.setFunction( func);
    df = null;
}

// Defines the initial value for the search.

public void setStartingValue( double start)
{
    result = start;
}
}
```

## 5.4 Example of Zero Finding—Roots of Polynomials

The zeroes of a polynomial function are called the *roots* of the polynomial. A polynomial of degree $n$ has at most $n$ real roots. Some[5] of them may be complex, but these are not covered in this book.

If $x_0$ is a root of the polynomial $P(x)$, then $P(x)$ can be exactly divided by the polynomial $x - x_0$. In other words, a polynomial $P_1(x)$ exists such that

$$P(x) = (x - x_0) \cdot P_1(x). \tag{5.5}$$

Equation 5.5 also shows that all roots of $P_1(x)$ are also roots of $P(x)$. Thus, one can carry the search of the roots using recurrence. In practice, a loop is more efficient.[6] The process is repeated at most times and will be interrupted if a zero-finding step does not converge.

5. If the degree of the polynomial is odd, there is always at least one noncomplex root. Polynomials of even degree may have only complex roots and no real roots.

6. The overhead comes from allocating the structures needed by the method in each call.

One could use the division algorithm of Section 2.2.1 to find $P_1(x)$. In this case, however, the inner loop of the division algorithm—that is, the loop over the coefficients of the dividing polynomial—is not needed since the dividing polynomial has only two terms. In fact, one does not need to express $x - x_0$ at all as a polynomial. To carry out the division, one uses a specialized algorithm taking the root as the only argument. This specialized division algorithm is called *deflation* [Press *et al.*].

Polynomials are very smooth, so Newton's algorithm is quite efficient for finding the first root. To ensure the best accuracy for the deflation, it is recommended to find the root of smallest absolute value first. This works without additional effort since our implementation of Newton's algorithm uses zero at the starting point by default. At each step, the convergence of the zero finder is checked. If a root cannot be found, the process must be stopped. Otherwise, the root finding loop is terminated when the degree of the deflated polynomial becomes zero.

### 5.4.1 Roots of Polynomials—Smalltalk Implementation

Roots of a polynomial can be obtained as an `OrderedCollection`. For example, Code Example 5.5 retrieves the roots of the polynomial $x^3 - 2x^2 - 13x - 10$.

**Code Example 5.5**

```
(DhbPolynomial coefficients: #(-10 -13 -2 1)) roots
```

The methods needed to get the roots are shown in Listing 5.5.

The deflation algorithm is implemented in the method `deflateAt:` using the iterator method `collect:` (see Section ???). An instance variable is keeping track of the remainder of the division within the block closure used by the method `collect:`.

The roots are kept in an `OrderedCollection` object constructed in the method `roots:`. The size of the `OrderedCollection` is initialized to the maximum expected number of real roots. Since some of the roots may be complex, we are storing the roots in an `OrderedCollection`, instead of an `Array`, so that the number of real roots found can be easily obtained. This method takes as argument the desired precision used in the zero-finding algorithm. A method `roots` uses the default numerical machine precision as discussed in Section 1.4.

---

**Listing 5.5 Smalltalk implementation of finding the roots of a polynomial**

| | |
|---|---|
| *Class* | `DhbPolynomial` |
| *Subclass of* | `Object` |
| *Instance variable names:* | `coefficients` |

*Instance Methods*

**deflatedAt:** aNumber

```
| remainder next newCoefficients|
remainder := 0.
newCoefficients := coefficients collect:
                    [ :each |
                      next := remainder.
                      remainder := remainder * aNumber + each.
                      next].
^self class new: ( newCoefficients copyFrom: 2 to:
                                    newCoefficients size) reverse
```

**roots**

```
^self roots: DhbFloatingPointMachine new
                                        defaultNumericalPrecision
```

**roots:** aNumber

```
| pol roots x rootFinder |
rootFinder := DhbNewtonZeroFinder new.
rootFinder desiredPrecision: aNumber.
pol := self class new: ( coefficients reverse collect: [ :each |
                                                    each asFloat]).
roots := OrderedCollection new: self degree.
[ rootFinder setFunction: pol; setDerivative: pol derivative.
  x := rootFinder evaluate.
  rootFinder hasConverged
    ] whileTrue: [ roots add: x.
                   pol := pol deflatedAt: x.
                   pol degree > 0
                     ifFalse: [ ^roots].
                 ].
^roots
```

---

## 5.4.2 Roots of Polynomials—Java Implementation

Roots of a polynomial can be obtained as an array. For example, Code Example 5.6 retrieves the roots of the polynomial $x^3 - 2x^2 - 13x - 10$.

**Code Example 5.6**

```
Double coef = { -10, -13, -2, 1};
PolynomialFunction p = new PolynomialFunction( coef);
Double roots = p.roots();
```

Finding the roots of a polynomial in Java is implemented in three methods—`deflate` and `roots` (two versions of this one)—of listing 2.5. The implementation follows that of Smalltalk.

Since the number of roots in a polynomial is not known at the beginning of the method, roots are kept in a Java `Vector` object. A `Vector` object is an array, that can grow in size. The manipulation of `Vector` objects, however, differs significantly from that of an array. First, the elements of a `Vector` object are instances of class `Object`. Thus, using the elements always requires casting into the proper class. The roots are primitive types (double) so they must first be cast into the corresponding wrapper class `Double`. Looping over the elements of a `Vector` object must be made using an enumeration instead of the conventional `for` loop.

## 5.5 Which Method to Choose?

There are other zero-finding techniques: *regula falsi*, Brent's method [Press *et al.*]. For each of these methods, however, a specialist of numerical methods can design a function causing that particular method to fail.

In practice, the bisection algorithm is quite slow, as can be seen from equation 5.2. Newton's algorithm is faster for most functions you will encounter. For example, it takes 5 iterations to find the zero of the logarithm function with Newton's algorithm to a precision of $3 \cdot 10^{-9}$, whereas the bisection algorithm requires 29 to reach a similar precision. On the other hand, bisection is rock solid and will always converge over an interval where the function has no singularity. Thus, it can be used as a recovery when Newton's algorithm fails.

My own experience is that Newton's algorithm is quite robust and very fast. It should suffice in most cases. As we have seen, Newton's algorithm will fail if it encounters a value for which the derivative of the function is very small. In this case, the algorithm jumps far away from the solution. For these cases, the chances are that the bisection algorithm will find the solution, if there is any. Thus, combining Newton's algorithm with bisection is the best strategy if you need to design a foolproof algorithm.

Implementing an object combining both algorithms is left as an exercise to the reader. Here is a quick outline of the strategy to adopt. Newton's algorithm must be modified to keep track of values for which the function takes negative values and positive values—that is, the values $x_{\text{pos}}$ and $x_{\text{neg}}$—making sure that the value $|x_{\text{pos}} - x_{\text{neg}}|$ never increases. Then, at each step, one must check that the computed change does not cause the solution to jump outside of the interval defined by $x_{\text{pos}}$ and $x_{\text{neg}}$. If that is the case, Newton's algorithm must be interrupted for one step using the bisection algorithm.

CHAPTER 6

# Integration of Functions

*Les petits ruisseaux font les grandes rivières*[1]
—French proverb

Many functions are defined by an integral. For example, the three functions discussed in the last three sections of Chapter 2 were all defined by an integral. When no other method is available, the only way to compute such function is to evaluate the integral. Integrals are also useful in probability theory to compute the probability of obtaining a value over a given interval. (This aspect will be discussed in Chapter ???.) Finally, integrals come up in the computation of surfaces and of many physical quantities related to energy and power. For example, the power contained in an electromagnetic signal is proportional to the integral of the square of the signal's amplitude.

The French proverb quoted at the beginning of this chapter is here to remind people that an integral is defined formally as the infinite sum of infinitesimal quantities.

## 6.1 Introduction

Let us begin with a concrete example. This time we shall take a problem from Physics 101.

When light is transmitted through a narrow slit, it is diffracted. The intensity of the light transmitted at an angle $\vartheta$, $I(\vartheta)$, is given by:

$$I(\vartheta) = \frac{\sin^2 \vartheta}{\vartheta^2}. \tag{6.1}$$

If one wants to compute the fraction of light that is transmitted within the first diffraction peak, one must compute the expression:

1. Small streams build great rivers.

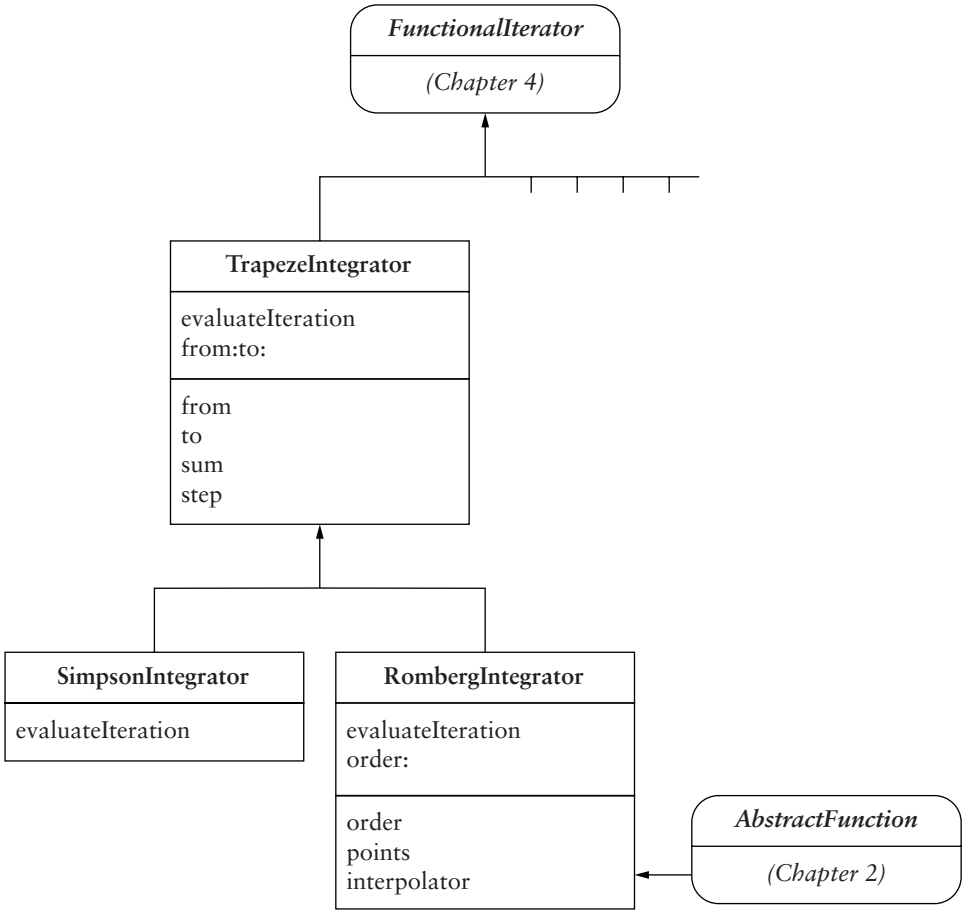


**FIG. 6.1** Class diagram of integration classes

$$I(\vartheta) = \frac{1}{\pi} \int_{-\pi}^{\pi} \frac{\sin^2 \vartheta}{\vartheta^2} d\vartheta. \tag{6.2}$$

The division by $\pi$ is there because the integral of $I(\vartheta)$ from $-\infty$ to $+\infty$ is equal to $\pi$. No closed form exists for the integral of equation 6.2: it must be computed numerically. This answer is 90.3%.

In this chapter, I introduce three integration algorithms. Figure 6.1 shows the corresponding class diagram.

The first one, trapeze integration, is only introduced for the sake of defining a common framework for the next two algorithms: Simpson and Romberg integration. In general, the reader should use Romberg's algorithm. It is fast and very precise. In some instances, however, Simpson's algorithm can be faster if high accuracy is not required.

## 6.2 General Framework—Trapeze Integration Method

Let me state at the beginning that one should not use the trapeze integration algorithm in practice. The interest of this algorithm is to define a general framework for numerical integration. All subclasses of the class responsible for implementing the trapeze integration algorithm will reuse most of the mechanisms described in this section.

The trapeze numerical integration method takes its origin in the series expansion of an integral. This series expansion is expressed by the Euler-Maclaurin formula [Bass]

$$\int_a^b f(x)\,dx = \frac{b-a}{2}\left[f(a)+f(b)\right] - \sum_n \frac{(b-a)^2}{(2n)!} B_{2n}\left[\frac{d^{2n-1}f(b)}{dx^{2n-1}} - \frac{d^{2n-1}f(a)}{dx^{2n-1}}\right], \tag{6.3}$$

where the numbers $B_{2n}$ are the Bernouilli numbers.

The next observation is that, if the interval of integration is small enough, the series in the second term of equation 6.2 would yield a contribution negligible compared to that of the first term. Thus, we can write:

$$\int_a^b f(x)\,dx \approx \frac{b-a}{2}\left[f(a)+f(b)\right], \tag{6.4}$$

if $b-a$ is sufficiently small. The approximation of equation 6.4 represents the area of a trapeze whose summits are the points circled in violet in figure 6.2. Finally, one must remember the additive formula between integrals:

$$\int_a^b f(x)\,dx = \int_a^c f(x)\,dx + \int_c^b f(x)\,dx, \tag{6.5}$$

for any $c$. We shall use this property by chosing a $c$ located between $a$ and $b$.

The resulting strategy is a divide-and-conquer strategy. The integration interval is divided until one can be sure that the second term of equation 6.2 becomes indeed negligible. As one would like to reuse the points at which the function has been evaluated during the course of the algorithm, the integration interval is halved at each iteration. The first few steps are outlined in Figure 6.2. An estimation of the integral is obtained by summing the areas of the trapezes corresponding to each partition.

Let $x_0^{(n)}, \ldots, x_{2^n}^{(n)}$ be the partition of the interval at iteration $n$. Let $\epsilon^{(n)}$ be the length of each interval between these points. We thus have:

$$\begin{cases} \epsilon^{(n)} = \frac{b-a}{2^n} \\ x_0^{(n)} = a \\ x_i^{(n)} = a + i\epsilon^{(n)} \quad \text{for } i = 1, \ldots, 2^n. \end{cases} \tag{6.6}$$

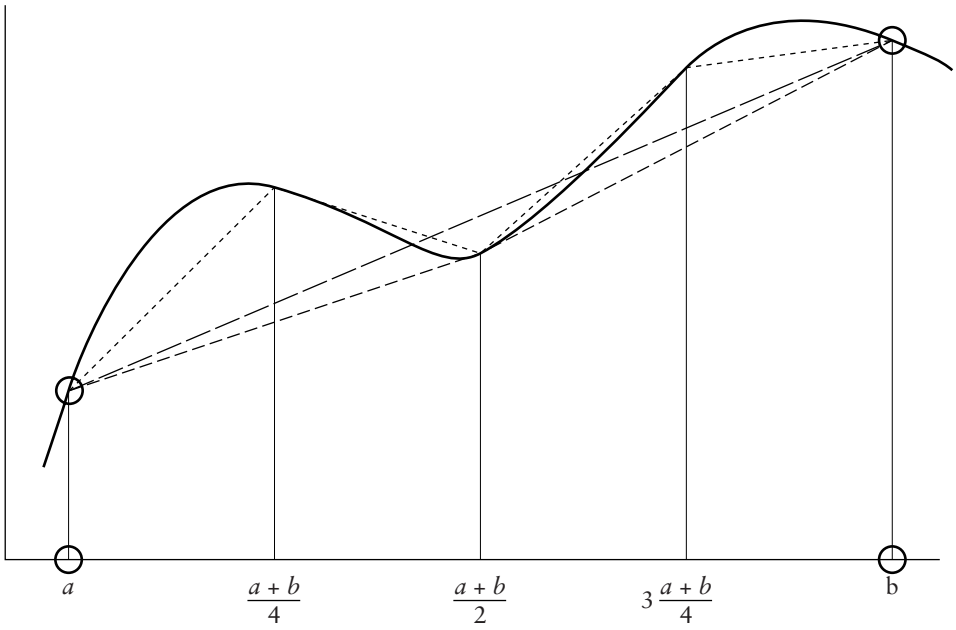


**FIG. 6.2** Geometrical interpretation of the trapeze integration method

The corresponding estimation for the integral is:

$$I^{(n)} = \epsilon^{(n)} \left[ f(a) + f(b) + 2 \sum_{i=1}^{2^n-1} f\left(x_i^{(n)}\right) \right]. \tag{6.7}$$

To compute the next estimation, it suffices to compute the value of the function at the even values of the partition because the odd values were already computed before. One can derive the following recurrence relation:

$$I^{(n+1)} = \frac{I^{(n)}}{2} + \epsilon^{(n)} \sum_{i=1}^{2^n-1} f\left(x_{2i-1}^{(n)}\right), \tag{6.8}$$

with the initial condition:

$$I^{(0)} = \frac{b-a}{2} \left[ f(a) + f(b) \right]. \tag{6.9}$$

Note that the sum on the right-hand side of equation 6.8 represents the sum of the function's values at the new points of the partition.

### 6.2.1 End Game Strategy

The final question is, When should the algorithm be stopped? A real honest answer is that we do not know. The magnitude of the series in equation 6.2 is difficult to estimate because the Bernouilli numbers become very large with increasing $n$. An experimental way is to watch for the change of the integral estimate. In other words, the absolute value of the last variation, $\left|I^{(n)} - I^{(n+1)}\right|$, is considered a good estimate of the precision. This kind of heuristic works for most functions.

At each iteration, the number of function evaluation doubles which means that the time spent in the algorithm grows exponentially with the number of iterations. Thus, the default maximum number of iteration must be kept quite low compared to that of the other methods.

In practice, however, trapeze integration converges quite slowly and should not be used. Why bother implementing it, then? It turns out that the more elaborate methods, Simpson and Romberg integration, require the computation of the same sums needed by the trapeze integration. Thus, the trapeze integration is introduced to be the superclass of the other, better integration methods.

One must keep in mind, however, that the magnitude of the series in equation 6.2 can become large for any function whose derivatives of high orders have singularities over the interval of integration. The convergence of the algorithm can be seriously compromised for such functions. This remark is true for the other algorithms described in this chapter. For example, none of the algorithms is able to give a precise estimate of the beta function using equation 2.32 with $x > 1$ and $y < 1$ (see Section 2.5) because, for these values, the derivatives of the function to integrate have a singularity at $t = 1$.

Another problem can come up if the function is nearly zeroes at regular intervals, for example, evaluating the integral of the function $f(x) = \frac{\sin(2^m x)}{x}$ from $-\pi$ to $\pi$ for a moderate value of $m$. In this case, the terms $I^{(0)}$ to $I^{(m)}$ will have a null contribution. This would cause the algorithm to stop prematurely. Such special function behavior is, of course, quite rare. Nevertheless, the reader must be aware of the limitations of the algorithm. This remark is valid for all algorithms exposed in this chapter.

### 6.2.2 Trapeze Integration—General Implementation

The class implementing trapeze integration is a subclass of the functional iterator discussed in Section 4.2. Two instance variables are needed to define the integration interval. Additional instance variables must keep track of the partition of the interval and of the successive estimations. Consequently, the class has the following instance variables.

`from` Contains the lower limit of the integration's interval (i.e., $a$).

`to` Contains the lower limit of the integration's interval (i.e., $b$).

`step` Contains the size of the interval's partition (i.e., $\epsilon^{(n)}$).

`sum` Contains the intermediate sums, (i.e., $I^{(n)}$).

Although trapeze integration is not a practical algorithm, I give an example of coding for both language implementations because the public interface used by trapeze integration is the same for all integration classes.

The example shown in the next two sections is the integration of the inverse function. In mathematics, the natural logarithm of $x$, $\ln x$, is defined as the integral from 1 to $x$ of the inverse function. Of course, using numerical integration is a very inefficient way of computing a logarithm. This example, however, allows the reader to investigate the accuracy (since the exact solution is known) and performances of all algorithms presented in this chapter. The interested reader should try the example for various settings of the desired precision and look at the number of iteration needed for a desired precision. He or she can also verify how accurate the estimated precision is.

### 6.2.3 Trapeze Integration—Smalltalk Implementation

Listing 6.1 shows the Smalltalk implementation of the trapeze integration method. In Smalltalk, the code for the computation of the integral defining the natural logarithm is as that given in Code Example 6.1.

**Code Example 6.1**

```
| integrator ln2 ln3|
integrator := DhbTrapezeIntegrator function:
[ :x | 1.0 / x] from: 1 to: 2.
ln2 := integrator evaluate.
integrator from: 1 to: 3.
ln3 := integrator evaluate.
```

The line after the declaration creates a new instance of the class `DhbTrapezeIntegrator` for the inverse function. The limits of the integration interval are set from 1 to 2 at creation time. The third line retrieves the value of the integral. The fourth line changes the integration interval, and the last line retrieves the value of the integral over the new integration interval.

The class `DhbTrapezeIntegrator` is a subclass of the class `AbstractFunctionIterator` defined in Section 4.2.1. The default creation class method `new` has been overloaded to prevent creating an object without initialized instance variables. The proper creation class method defines the function and the integration interval.

The method `from:to:` allows changing the integration interval for a new computation with the same function.

Note that the initialization of the iterations (method `computeInitialValues`; see Section 4.1.1) also corresponds to the first iteration of the algorithm. The method `highOrderSum` computes the sum of the right-hand side of equation 6.8.

**Listing 6.1** **Smalltalk implementation of trapeze integration**

| | |
|---|---|
| *Class* | `DhbTrapezeIntegrator` |
| *Subclass of* | `DhbFunctionalIterator` |
| *Instance variable names:* | `from to sum step` |

*Class Methods*

**defaultMaximumIterations**

```
^13
```

**new**

```
^self error: 'Method new:from:to: must be used'
```

**new:** aBlock **from:** aNumber1 **to:** aNumber2

```
^super new initialize: aBlock from: aNumber1 to: aNumber2
```

*Instance Methods*

**computeInitialValues**

```
step := to - from.
sum := ( ( functionBlock value: from) +
                                ( functionBlock value: to)) * step /2.
result := sum.
```

**evaluateIteration**

```
| oldResult |
oldResult := result.
result := self higherOrderSum.
^self relativePrecision: ( result - oldResult) abs
```

**from:** aNumber1 **to:** aNumber2

```
from := aNumber1.
to := aNumber2.
```

**higherOrderSum**

```
| x newSum |
x := step / 2 + from.
newSum := 0.
[ x < to ]
```

```
        whileTrue: [ newSum := ( functionBlock value: x) + newSum.
                     x := x + step.
                   ].
    sum := ( step * newSum + sum) / 2.
    step := step / 2.
    ^sum
```

**initialize:** `aBlock` **from:** `aNumber1` **to:** `aNumber2`

```
    functionBlock := aBlock.
    self from: aNumber1 to: aNumber2.
    ^self
```

---

### 6.2.4 Trapeze Integration—Java Implementation

Listing 6.2 shows the Java implementation of the trapeze integration method. Code Example 6.2 gives the code for computing the integral defining the natural logarithm in Java.

**Code Example 6.2**

```
TrapezeIntegrator integrator = new TrapezeIntegrator( new
       OneVariableFunction() { public double value( double x)
       { return 1/x;}}, 1, 2);
integrator evaluate();
double ln2 = integrator.getResult();
integrator.setInterval( 2, 3);
double ln3 = integrator.getResult();
```

The first line creates a new instance of the class `TrapezeIntegrator` for the inverse function defined as an inner class. The limits of the integration interval are set from 1 to 2. The second line retrieves the value of the integral. The third line changes the integration interval, and the last line retrieves the value of the integral over the new integration interval. Required checks for convergence are not included in this example.

The class is a subclass of the class `FunctionalIterator` defined in Section 4.2.2. A single constructor method creates a new instance with a function and the integration interval. The method `setInterval` allows changing the integration interval for a new computation with the same function.

Note that the initialization of the iterations (method `initializeIterations`; see Section 4.1.2) also corresponds to the first iteration of the algorithm. The method `highOrderSum` computes the sum of the right-hand side of equation 6.8.

**Listing 6.2 Java implementation of trapeze integration**

```
package DhbIterations;

import DhbInterfaces.OneVariableFunction;

// Trapeze integration method

// @author Didier H. Besset

public class TrapezeIntegrator extends FunctionalIterator
{

    // Low integral bound.

    private double from;

    // High integral bound.

    private double to;

    // Sum

   protected double sum;

    // Interval partition.

    private double step;

// Constructor
// @param func DhbInterfaces.OneVariableFunction
// @param from double
// @param to double

public TrapezeIntegrator(OneVariableFunction f, double from, double to)
{
    super( f);
    setInterval( from, to);
    setMaximumIterations( 13);
}
public double evaluateIteration()
{
    double oldResult = result;
    result = highOrderSum();
    return relativePrecision( Math.abs( result - oldResult));
}
```

```
// @return double

protected double highOrderSum()
{
    double x = from + 0.5 * step;
    double newSum = 0;
    while ( x < to )
    {
        newSum += f.value(x);
        x += step;
    }
    sum = ( step * newSum + sum) * 0.5;
    step *= 0.5;
    return sum;
}
public void initializeIterations()
{
    step = to - from;
    sum = ( f.value(from) + f.value(to)) * step * 0.5;
    result = sum;
}

// Defines integration interval.
// @param double a low integral bound.
// @param double b high integral bound.

public void setInterval( double a, double b)
{
    from = a;
    to = b;
}
}
```

## 6.3 Simpson Integration Algorithm

Simpson integration algorithm consists in replacing the function to integrate by a second-order Lagrange interpolation polynomial [Bass] (see Section 3.2). One can then carry the integral analytically. Let $f(x)$ be the function to integrate. For a given interval of integration, $[a, b]$, the function is evaluated at the extremities of the interval and at its middle point $c = \frac{a+b}{2}$. As defined in equation 3.1, the second-order Lagrange interpolation polynomial is then given by:

$$P_2(x) = \frac{2}{(b-a)^2}\left[(x-b)(x-c)f(a) + (x-c)(x-a)f(b) + (x-a)(x-b)f(c)\right]. \tag{6.10}$$

The integral of the polynomial over the interpolation interval is given by:

$$\int_a^b P_2(x)dx = \frac{b-a}{6}\left[f(a)+f(b)+4f(c)\right]. \tag{6.11}$$

As for trapeze integration, the interval is partitioned into small intervals. Let us assume that the interval has been divided into subintervals. By repeating equation 6.11 over each subinterval, we obtain

$$\int_a^b P_2(x)dx = \frac{\epsilon^{(n)}}{3}\left[f(a)+f(b)+2\sum_{i=1}^{2^{n-1}} f(x_{2i-1}^{(n)})+4\sum_{i=0}^{2^{n-1}} f(x_{2i}^{(n)})\right]. \tag{6.12}$$

Equation 6.12 uses the notations introduced in Section 6.2. Except for the first iteration, the right-hand side of equation 6.12 can be computed from the quantities $I^{(n)}$ defined in equation 6.7. Thus, we have

$$\int_a^b P_2(x)dx = \frac{1}{3}\left[4I^{(n)} - I^{(n-1)}\right] \quad \text{for } n > 1. \tag{6.13}$$

This can be checked by verifying that $I^{(n-1)}$ is equal to the first sum of equation 6.12 times and that $I^{(n)}$ is equal to the addition of the two sums of equation 6.12 times $\epsilon^{(n)}$. As noted in Section 6.2, we can reuse the major parts of the trapeze algorithm: computation of the sums and partition of the integration interval.

Like in the case of the trapeze algorithm, the precision of the algorithm is estimated by looking at the differences between the estimation obtained previously and the current estimation. At the first iteration, only one function point is computed. This can cause the process to stop prematurely if the function is nearly zero at the middle of the integration interval. Thus, a protection must be built in to prevent the algorithm from stopping at the first iteration.

### 6.3.1 Simpson Integration—General Implementation

The class implementing Simpson algorithm is a subclass of the class implementing trapeze integration. The method `evaluateIteration` is the only method needing change. The number of iterations is checked to prevent returning after the first iteration.

The public interface is the same as that of the superclass. Thus, all the examples shown in Sections 6.2.3 and 6.2.4 can be used for Simpson algorithm by just changing the name of the class.

### 6.3.2 Simpson Integration—Smalltalk Implementation

Listing 6.3 shows the complete implementation in Smalltalk.

The class `DhbSimpsonIntegrator` is a subclass of the class `DhbTrapezeIntegrator` defined in Section 6.2.3.

**Listing 6.3** **Smalltalk implementation of the Simpson integration algorithm**

*Class* DhbSimpsonIntegrator
*Subclass of* DhbTrapezeIntegrator

***Instance Methods***

**evaluateIteration**

```
| oldResult oldSum |
iterations < 2
    ifTrue: [ self higherOrderSum.
              ^1
            ].
oldResult := result.
oldSum := sum.
result := (self higherOrderSum * 4 - oldSum) / 3.
^self relativePrecision: ( result - oldResult) abs
```

## 6.3.3 Simpson Integration—Java Implementation

Listing 6.4 shows the complete implementation in Java.

Except for the need of an explicit constructor method, the Java implementation is exactly similar to that in Smalltalk.

**Listing 6.4** **Java implementation of the Simpson integration algorithm**

```
package DhbIterations;


// Simpson integration method

// @author Didier H. Besset

public class SimpsonIntegrator extends TrapezeIntegrator
{

// SimpsonIntegrator constructor.
// @param f DhbInterfaces.OneVariableFunction
// @param from double
// @param to double

public SimpsonIntegrator(DhbInterfaces.OneVariableFunction f,
                                              double from, double to)
{
```

```
        super(f, from, to);
    }

    // @return double

    public double evaluateIteration()
    {
        if ( getIterations() < 2 )
        {
            highOrderSum();
            return getDesiredPrecision( );
        }
        double oldResult = result;
        double oldSum = sum;
        result = ( 4 * highOrderSum() - oldSum) / 3.0;
        return relativePrecision( Math.abs( result - oldResult));
    }
    }
```

## 6.4 Romberg Integration Algorithm

If one goes back to equation 6.2, one notices that the second term is of the order of the square of the integration interval. Romberg's algorithm uses this fact to postulate that $I^{(n)}$ is a smooth function of the square of the size of interval's partition $\epsilon^{(n)}$. Romberg's algorithm introduces the parameter $k$, where $k-1$ is the degree of the interpolation's polygon.[2] The result of the integral is estimated by extrapolating the series $I^{(n-k)}, \ldots, I^{(n)}$ at the value $\epsilon^{(n)} = 0$. Since we have

$$\epsilon^{(n)} = \frac{\epsilon^{(n-1)}}{2}, \tag{6.14}$$

one only needs to interpolate over successive powers of 1/4 , starting with 1: 1, 1/4, 1/16, 1/256, and so forth. In this case, extrapolation is safe because the value at which extrapolation is made is very close to the end of the interval defined by the sample points and actually becomes closer and closer with every iteration.

Thus, Romberg's algorithm requires at least $k$ iterations. A polynomial of fourth degree—that is, $k = 5$—is generally sufficient [Press *et al.*]. The good news is that this algorithm converges very quickly, and, in general, only a few iterations are needed after the five initial ones.

2. In other words, $k$ is the number of points over which the interpolation is performed (see Section 3.2).

Extrapolation is performed using Neville's algorithm (see Section 3.4) because it computes an estimate of the error on the interpolated value. That error estimate can then be used as the estimate of the error on the final result.

If $k = 1$, Romberg's algorithm is equivalent to trapeze integration. If $k = 2$, the interpolation polynomial is given by

$$P_1(x) = y_1 + \frac{x - x_1}{x_2 - x_1}(y_2 - y_1). \tag{6.15}$$

At the $n$th iteration we have $y_1 = I^{(n-1)}$, $y_2 = I^{(n)}$, and $x_2 = x_1/4$. Thus, the interpolated value at zero is:

$$P_1(0) = I^{(n-1)} + \frac{4}{3}\left[I^{(n)} - I^{(n-1)}\right] = \frac{1}{3}\left[4I^{(n)} - I^{(n-1)}\right]. \tag{6.16}$$

Thus, for $k = 2$ Romberg's algorithm is equivalent to Simpson's algorithm. For higher order, however, Romberg's algorithm is much more efficient than Simpson method as soon as precision is required (see a comparison of the result in Section 6.6).

Using interpolation on the successive results of an iterative process to obtain the final result is a general procedure known as Richardson's deferred approach to the limit [Press *et al.*]. This technique can be used whenever the estimated error can be expressed as a function of a suitable parameter depending on the iterations. The choice of the parameter, however, is critical. For example, if one had interpolated over the size of the interval's partition instead of its square, the method would not converge as well.[3]

### 6.4.1 Romberg Integration—General Implementation

The class implementing Romberg's algorithm needs the following additional instance variables:

`order` The order of the interpolation, (i.e., $k$),

`interpolator` An instance of Neville's interpolation class

`points` An `OrderedCollection` containing the most recent sum estimates, (i.e., $I^{(n-k)}, \ldots, I^{(n)}$).

The method `evaluateIteration` (see Section 4.1) contains the entire algorithm. At each iteration, the collection of points receives a new point with an abscissa equal to the quarter of that of the last point and an ordinate equal to the next sum estimate. If not enough points are available, the method returns a precision such that the iterative process will continue. Otherwise, the extrapolation is performed. After the result of

3. It converges at the same speed as Simpson's algorithm. This can be verified by running the comparison programs after changing the factor 0.25 used to compute the abscissa of the next point into 0.5.

the extrapolation has been obtained, the oldest point is removed. In other words, the collection of points is used as a last in, last out list with a constant number of elements equal to the order of the interpolation. Of the two values returned by Neville's interpolation (see Section 3.4), the interpolated value is stored in the result, and the error estimate is returned as the precision for the other.

### 6.4.2 Romberg Integration—Smalltalk Implementation

Listing 6.5 shows the Smalltalk implementation of Romberg's algorithm. The class `DhbRombergIntegrator` is a subclass of the class `DhbTrapezeIntegrator` defined in Section 6.2.3.

The class method `defaultOrder` defines the default order to 5. This method is used in the method `initialize` so that each newly created instance is created with the default interpolation order. The method `order:` allows changing the default order if needed.

The sample points defining the interpolation are stored in an `OrderedCollection`. This collection is created in the method `computeInitialValues`. Since the number of points will never exceed the order of the interpolation, the maximum size is preset when the collection is created. The method `computeInitialValues` also creates the object in charge of performing the interpolation, and it stores the first point in the collection of sample points.

---

**Listing 6.5 Smalltalk implementation of Romberg integration**

| | |
|---|---|
| *Class* | `DhbRombergIntegrator` |
| *Subclass of* | `DhbTrapezeIntegrator` |
| *Instance variable names:* | `order points interpolator` |

*Class Methods*

**defaultOrder**

```
^5
```

*Instance Methods*

**computeInitialValues**

```
super computeInitialValues.
points := OrderedCollection new: order.
interpolator := DhbNevilleInterpolator points: points.
points add: 1 @ sum.
```

**evaluateIteration**

```
| interpolation |
```

```
    points addLast: (points last x * 0.25) @ self higherOrderSum.
    points size < order
        ifTrue: [ ^1].
    interpolation := interpolator valueAndError: 0.
    points removeFirst.
    result := interpolation at: 1.
    ^self relativePrecision: ( interpolation at: 2) abs
```

**initialize**

```
    order := self class defaultOrder.
    ^super initialize
```

**order:** anInteger

```
    anInteger < 2
        ifTrue: [ self error: 'Order for Romberg integration must be
                                                        larger than 1'].
    order := anInteger.
```

---

## 6.4.3 Romberg Integration—Java Implementation

Listing 6.6 shows the Java implementation of Romberg's algorithm. The class `DhbRombergIntegrator` is a subclass of the class `TrapezeIntegrator` defined in Section 6.2.4.

The default order is assigned in the instance variable so that each newly created instance is created with the default interpolation order (5). The method `setOrder` allows changing the default order when needed.

The sample points defining the interpolation are stored in a structure implementing the `PointSeries` interface as discussed in Section 3.2.2. The method `initializeIterations` creates this structure. It also creates the interpolator object in charge of performing the interpolation, and it stores the first point in the collection of sample points.

---

**Listing 6.6 Java implementation of Romberg integration**

```
package DhbIterations;

import DhbInterfaces.OneVariableFunction;
import DhbInterpolation.NevilleInterpolator;
import DhbScientificCurves.Curve;

// Romberg integration method

// @author Didier H. Besset
```

```
public class RombergIntegrator extends TrapezeIntegrator
{

    // Order of the interpolation.

    private int order = 5;

    // Structure containing the last estimations.

    private Curve estimates;

    // Neville interpolator.

    private NevilleInterpolator interpolator;

// RombergIntegrator constructor.
// @param func DhbInterfaces.OneVariableFunction
// @param from double
// @param to double

public RombergIntegrator(DhbInterfaces.OneVariableFunction func,
                                            double from, double to)
{
    super(func, from, to);
}

// @return double

public double evaluateIteration()
{
    estimates.addPoint( estimates.xValueAt(estimates.size() - 1) * 0.25,
                                                    highOrderSum());
    if ( estimates.size() < order )
        return 1;
    double[] interpolation = interpolator.valueAndError( 0);
    estimates.removePointAt( 0);
    result = interpolation[0];
    return relativePrecision( Math.abs( interpolation[1]));
}
public void initializeIterations()
{
    super.initializeIterations();
    estimates = new Curve();
    interpolator = new NevilleInterpolator( estimates);
    estimates.addPoint( 1, sum);
}
```

```
// @param n int

public void setOrder( int n)
{
    order = n;
}
}
```

## 6.5 Evaluation of Open Integrals

An *open integral* is an integral for which the function to integrate cannot be evaluated at the boundaries of the integration interval. This is the case when one of the limits is infinite or when the function to integrate has a singularity at one of the limits. If the function to integrate has one or more singularity in the middle of the integration interval, the case can be reduced to that of having a singularity at the limits using the additive formula between integrals 6.5. Generalization of the trapeze algorithm and the corresponding adaptation of Romberg's algorithm can be found in [Press *et al.*].

### 6.5.1 Bag of tricks

My experience is that using a suitable change of variable can often remove the problem. In particular, integrals whose integration interval extends to infinity can be rewritten as integrals over a finite interval. I give a few examples here.

For an integral starting from minus infinity, a change of variable $t = \frac{1}{x}$ can be used as follows:

$$\int_{-\infty}^{a} f(x)dx = \int_{\frac{1}{a}}^{0} f\left(\frac{1}{t}\right)\frac{dt}{t^2} \quad \text{for } a < 0. \tag{6.17}$$

For such an integral to be defined, the function must vanish at minus infinity faster than $x^2$. This means that

$$\lim_{t\to 0} \frac{1}{t^2} f(\frac{1}{t}) = 0. \tag{6.18}$$

If $a > 0$, the integration must be evaluated in two steps—for example, one over the interval $[-\infty, -1]$ using the change of variable of equation 6.17, and one over the interval $[-1, a]$ using the original function.

For an integral ending at positive infinity, the same change of variable can also be made. However, if the interval of integration is positive, the change of variable $t = e^{-x}$ can be more efficient. In this case, one makes the following transformation:

$$\int_{a}^{+\infty} f(x)dx = \int_{0}^{e^{-a}} f(-\ln t)\frac{dt}{t} \quad \text{for } a > 0. \tag{6.19}$$

For this integral to be defined, one must have

$$\lim_{t \to 0} \frac{1}{t} f(\ln t) = 0. \tag{6.20}$$

By breaking up the interval of integration into several pieces, one can choose a change of variable most appropriate for each piece.

## 6.6 Which Method to Chose?

An example comparing the results of the three algorithms is given in Section 6.6.1 for Smalltalk and Section 6.6.2 for Java. The function to integrate is the inverse function in both cases. The integration interval is from 1 to 2 so that the value of the result is known, namely ln 2. Integration is carried for various values of the desired precision. The reader can then compare the attained precision (both predicted and real) and the number of iterations[4] required for each algorithm. Recall that the number of required function evaluations grows exponentially with the number of iterations.

The results clearly show that the trapeze algorithm is ruled out as a practical method. As noted in Section 6.2, it is not converging quickly toward a solution.

Romberg's algorithm is the clear winner. At a given precision, it requires the least number of iterations. This is the algorithm of choice in most cases.

Simspon's algorithm may be useful if the required precision is not too high and if the time to evaluate the function is small compared to the interpolation. In such cases, Simspon's algorithm can be faster than Romberg's algorithm.

Sections 6.6.1 and 6.6.2 gives some sample code the reader can use to investigate the various integration algorithms. The results of the code execution are shown in Table 6.1. The columns of this table are as follows:

$\epsilon_{\max}$ The desired precision

$n$ The number of required iterations; let us recall that the corresponding number of function's evaluations is $2^{n+1}$

$\tilde{\epsilon}$ The estimated precision of the result

$\epsilon$ The effective precision of the result—that is, the absolute value of the difference between the result of the integration algorithm and the true result

### 6.6.1 Smalltalk Comparison

The script of Listing 6.7 can be executed as such in any Smalltalk window.

The function to integrate is specified as a block closure as discussed in Section 2.1.1.

4. Note that the number of iterations printed in the examples is one less than the real number of iterations because the first iteration is performed in the setup phase of the iterative process.

**TABLE 6.1** Comparison between integration algorithms

| | Trapeze algorithm | | | Simpson algorithm | | | Romberg algorithm | | |
|---|---|---|---|---|---|---|---|---|---|
| $\epsilon_{max}$ | $n$ | $\tilde{\epsilon}$ | $\epsilon$ | $n$ | $\tilde{\epsilon}$ | $\epsilon$ | $n$ | $\tilde{\epsilon}$ | $\epsilon$ |
| $10^{-5}$ | 8 | $4.1 \cdot 10^{-6}$ | $9.5 \cdot 10^{-7}$ | 4 | $9.9 \cdot 10^{-6}$ | $4.7 \cdot 10^{-7}$ | 4 | $1.7 \cdot 10^{-9}$ | $1.4 \cdot 10^{-9}$ |
| $10^{-7}$ | 11 | $6.4 \cdot 10^{-8}$ | $1.5 \cdot 10^{-8}$ | 6 | $4.0 \cdot 10^{-8}$ | $1.9 \cdot 10^{-9}$ | 4 | $1.7 \cdot 10^{-9}$ | $1.4 \cdot 10^{-9}$ |
| $10^{-9}$ | 15 | $2.5 \cdot 10^{-10}$ | $5.8 \cdot 10^{-11}$ | 8 | $1.5 \cdot 10^{-10}$ | $7.3 \cdot 10^{-12}$ | 5 | $1.4 \cdot 10^{-11}$ | $3.7 \cdot 10^{-12}$ |
| $10^{-11}$ | 18 | $3.9 \cdot 10^{-12}$ | $9.0 \cdot 10^{-13}$ | 9 | $9.8 \cdot 10^{-12}$ | $5.7 \cdot 10^{-13}$ | 6 | $7.6 \cdot 10^{-14}$ | $5.7 \cdot 10^{-15}$ |
| $10^{-13}$ | 21 | $4.8 \cdot 10^{-14}$ | $2.8 \cdot 10^{-14}$ | 11 | $3.8 \cdot 10^{-14}$ | $1.9 \cdot 10^{-15}$ | 6 | $7.6 \cdot 10^{-14}$ | $5.7 \cdot 10^{-15}$ |

**Listing 6.7 Smalltalk comparison script for integration algorithms**

```
| a b  integrators |
a := 1.0.
b := 2.0.
integrators := Array with: ( DhbTrapezeIntegrator new: [ :x | 1.0 / x]
from: a to: b)
with: ( DhbSimpsonIntegrator new: [ :x | 1.0 / x] from: a to: b)
with: ( DhbRombergIntegrator new: [ :x | 1.0 / x] from: a to: b).
#(1.0e-5 1.0e-7 1.0e-9 1.0e-11 1.0e-13) do: [ :precision |
Transcript cr; cr; nextPutAll: '===> Precision: '.
precision printOn: Transcript.
integrators do: [ :integrator |
Transcript cr; nextPutAll: '*****   ', integrator class name,':'; cr.
integrator desiredPrecision: precision.
Transcript nextPutAll: 'Integral of 1/x from '.
a printOn: Transcript.
Transcript nextPutAll: ' to '.
b printOn: Transcript.
Transcript nextPutAll: ' = '.
integrator evaluate printOn: Transcript.
Transcript nextPutAll: ' +- '.
integrator precision printOn: Transcript.
Transcript cr; nextPutAll: ' ( '.
integrator iterations printOn: Transcript.
Transcript nextPutAll: ' iterations, true error = '.
( integrator result - 2 ln) printOn: Transcript.
Transcript nextPutAll: ')'; cr.
]]
```

### 6.6.2 Java Comparison

The method shown in Listing 6.8 can be compiled as a static method in any class. To execute the comparison test, this method should be called from a main method with the desired precision. The function to integrate is implemented as an inner class as discussed in Section 2.1.2.

**Listing 6.8 Java comparison method for integration algorithms**

```
public static void executeAtPrecision( double precision)
{
    double a = 1;
    double b = 2;
    System.out.println(" ");
    System.out.println("====> desired precision = " + precision);
    System.out.println("*****   Trapeze method:");
    try {
            TrapezeIntegrator integratorT = new TrapezeIntegrator(
                                            new OneVariableFunction()
                { public double value( double x){ return 1/x;}}, a, b);
            integratorT.setDesiredPrecision( precision);
            integratorT.setMaximumIterations( 25);
            integratorT.evaluate();
            System.out.print("Integral of ln(x) from "+a+" to "+b);
            System.out.println(" is "+integratorT.getResult()+" +/- "
                                        + integratorT.getPrecision());
            System.out.print(" ( "+integratorT.getIterations()
                                        +" iterations");
            System.out.println(", error = "+(Math.log( b)
                                - integratorT.getResult())+")");
        } catch ( IllegalArgumentException e)
        { System.out.println("Illegal precision specified
                                         in integration");};
    System.out.println("*****   Simpson method:");
    try {
            SimpsonIntegrator integratorS = new SimpsonIntegrator(
                                            new OneVariableFunction()
               { public double value( double x){ return 1/x;}}, a, b);
            integratorS.setDesiredPrecision( precision);
            integratorS.evaluate();
            System.out.print("Integral of ln(x) from "+a+" to "+b);
            System.out.println(" is "+integratorS.getResult()+" +/- "
                                    +integratorS.getPrecision());
            System.out.print(" ( "+integratorS.getIterations()
                                +" iterations");
            System.out.println(", error = "+(Math.log( b)
```

```
                                        - integratorS.getResult())+")");
            } catch ( IllegalArgumentException e)
            { System.out.println("Illegal precision specified in
                                 integration");};
        System.out.println("*****   Romberg method:");
        try {
                RombergIntegrator integratorR = new RombergIntegrator(
                                                new OneVariableFunction()
                    { public double value( double x){ return 1/x;}}, a, b);
                integratorR.setDesiredPrecision( precision);
                integratorR.evaluate();
                System.out.print("Integral of ln(x) from "+a+" to "+b);
                System.out.println(" is "+integratorR.getResult()+" +/- "
                                        +integratorR.getPrecision());
                System.out.print(" ( "+integratorR.getIterations()
                                             +" iterations");
                System.out.println(", error = "+(Math.log( b)
                                    - integratorR.getResult())+")");
            } catch ( IllegalArgumentException e)
            { System.out.println("Illegal precision specified
                                 in integration");};
    }
```

# References


[Abramovitz & Stegun] Milton Abramovitz and Irene A. Stegun, *Handbook of Mathematical Functions*, Dover, 1964.

[Achtley & Bryant] William R. Achtley and Edwin H. Bryant editors, *Benchmark Papers in Systematic and Evolutionary Biology*, Vol. 1, Dowden, Hutchinson & Ross, Inc., Stroudsburg, Pa.; distributed by Halsted Press [John Wiley & Sons, Inc.], New York, 1975.

[Bass] J. Bass, *Cours de Mathématiques*, Tome II, Masson, 1968.

[Beck] Kent Beck, *Smalltalk Best Practice Patterns*, Prentice Hall, 1997.

[Berry & Linoff] Michael J. A. Berry and Gordon Linoff, *Data Mining for Marketing, Sales and Customer Support*, Wiley, 1997.

[Cormen *et al.*] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, *Introduction to Algorithms*, McGraw-Hill, 1990.

[Gamma *et al.*] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns*, Addison-Wesley, 1995.

[Gullberg] Jan Gullberg, *Mathematics from the Birth of the Numbers*, Norton, 1997.

[Ifrah] Georges Ifrah, *Histoire Universelle des Chiffres*, Robert Laffont, 1994.

[Knudth 1] Donald E. Knuth, *The Art of Computer Programming*, Vol. 1, Addison-Wesley, 1973.

[Knudth 2] Donald E. Knuth, *The Art of Computer Programming*, Vol. 2, Addison-Wesley, 1981.

[Knudth 3] Donald E. Knuth, *The Art of Computer Programming*, Vol. 3, Addison-Wesley, 1973.

[Koza *et al.*] John R. Koza, Forrest H, Bennett III, David Andre, and Martin A. Keane, *Genetic Programming III*, Morgan Kaufmann, 1999.

[Law & Kelton] Averill M. Law and W. David Kelton, *Simulation Modeling and Analysis*, McGraw-Hill, 1982.

[Phillips & Taylor] G. M. Phillips and P. J. Taylor, *Theory and Applications of Numerical Analysis*, Academic Press, 1973.

[Press *et al.*] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, *Numerical Recipes for C: The Art of Scientific Computing*, Cambridge University Press, 1992.

[Alpert *et al.*] Sherman R. Alpert, Kyle Brown, and Bobby Woolf, *Design Pattern Smalltalk Companion*, Addison-Wesley, 1998.

[Smith] David N. Smith, *IBM Smalltalk: The Language*, Addison-Wesley, 1995.

[Flanagan] David Flanagan, *Java in a Nutshell*, O'Reilly, 1996.