PolyMath

Serge Stinckwich, Stéphane Ducasse

December 31, 2017
commit 945cace*

[&] Developed for the books published by Square Bracket Associates.
http://squarebracketassociates.org

Contents

Illustrations

About this version

We would like to thank Didier Besset for his great book and for his gift of the source and implementation to the community.

This is an abridged version of Didier's book, without the Java implementation and reference; our goal is to make the book slimmer and easier to read. The implementation presented in this book is part of the Polymath library. Both versions of the book are now maintained under open-source terms and are available at the following URLs:

- Abridged version (this book)
  https://github.com/SquareBracketAssociates/NumericalMethods
- Archive of the original book, with code in both Java and Smalltalk
  https://github.com/SquareBracketAssociates/ArchiveOONumericalMethods
- PolyMath library https://github.com/PolyMathOrg/PolyMath

Both this and the full version are maintained by Stéphane Ducasse and Serge Stinckwich. Remember that we are all Charlie.

7 December 2016

Preface

Si je savais une chose utile à ma nation qui fût ruineuse à une autre,
je ne la proposerais pas à mon prince,
parce que je suis homme avant d'être Français,
parce que je suis nécessairement homme,
et que je ne suis Français que par hasard.[1]
Charles de Montesquieu

When I first encountered object-oriented programming I immediately became highly enthusiastic about it, mainly because of my mathematical inclination. After all I learned to use computers as a high-energy physicist. In mathematics, a new, high order concept is always based on previously defined, simpler, concepts. Once a property is demonstrated for a given concept it can be applied to any new concept sharing the same premises as the original one. In object-oriented language, this is called reuse and inheritance. Thus, numerical algorithms using mathematical concepts that can be mapped directly into objects.

This book is intended to be read by object-oriented programmers who need to implement numerical methods in their applications. The algorithms exposed here are mostly fundamental numerical algorithms with a few advanced ones. The purpose of the book is to show that implementing these algorithms in an object-oriented language is feasible and quite easily feasible. We expect readers to be able to implement their own favorite numerical algorithm after seeing the examples discussed in this book.

The scope of the book is limited. It is not a Bible about numerical algorithms. Such Bible-like books already exist and are quoted throughout the chapters. Instead I wanted to illustrate mapping between mathematical concepts and computer objects. I have limited the book to algorithms, which I have implemented and used in real applications over twelve years of object-oriented programming. Thus, the reader can be certain that the algorithms have been tested in the field.

[1] If I knew some trade useful to my country, but which would ruin another, I would not disclose it to my ruler, because I am a man before being French, because I belong to mankind while I am French only by a twist of fate.

Because the book's intent is to show numerical methods to object- oriented programmers, the code presented here is described in depth. Each algorithm is presented with the same organization. First the necessary equations are introduced with short explanations. This book is not one about mathematics so explanations are kept to a minimum. Then the general object-oriented architecture of the algorithm is presented. Finally, this book is intented to be a practical one, the code implementation is exposed. First, I describe how to use it, for readers who are just interested in using the code directly and then I discss and present the code implementation.

As far as possible each algorithm is presented with examples of use. I did not want to build contrived examples and instead have used examples personally encountered in my professional life. Some people may think that some examples are coming from esoteric domains. This is not so. Each example has been selected for its generality. The reader should study each example regardless of the field of application and concentrate on the universal aspects of it.

Acknowledgements

The author wishes to express his thanks to the many people with whom he had interactions about the object-oriented approach — Smalltalk and Java in particular — on the various electronic forums. One special person is Kent Beck whose controversial statements raised hell and started spirited discussions. I also thank Kent for showing me tricks about the Refactoring Browser and *e*Xtreme Programming. I also would like to thank Eric Clayberg for pulling me out of a ditch more than once and for making such fantastic Smalltalk tools.

A special mention goes to Prof. Donald Knuth for being an inspiration for me and many other programmers with his series of books *T*he Art of Computer Programming, and for making this wonderful typesetting program TEX. This present book was typeset with TEX and LATEX.

Furthermore, I would like to give credit to a few people without whom this present book would never have been published. First, Joseph Pelrine who persuaded me that what I was doing was worth sharing with the rest of the object-oriented community.

The author expresses his most sincere thanks to the reviewers who toiled on the early manuscripts of this book. Without their open-mindedness this book would never had made it to a publisher.

Special thanks go to David N. Smith for triggering interesting thoughts about random number generators and to Dr. William Leo for checking the equations.

Finally my immense gratitude is due to Dr. Stéphane Ducasse of the University of Bern who checked the orthodoxy of the Smalltalk code and who did

a terrific job of rendering the early manuscript not only readable but entertaining.

Genolier, 11 April 2000

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 only used by a very small amount of 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 getting commonplace. The wealth of information stored in large databases tends to be ignored, mainly because only few persons knows how to get access to it and an even fewer number know how to extract useful information. Recently people have started to tackle this problem under the buzzword data mining. In truth, data mining is nothing else than good old numerical data analysis performed by high-energy physicists with the help of computers. Of course a few new techniques are 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.

[1] Science without consciousness just ruins the soul.

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

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 guide goes further in stating that *a* model is a simplification of reality and we should emphasize that it is *o*nly 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 where the behavior of computer objects nearly reproduces that of their real-world counterparts is mathematics. Mathematical objects are organized within *h*ierarchies. For example, natural integers are included in integers (signed integers), which are included in rational numbers, themselves included in real numbers. Mathematical objects use *p*olymorphism 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 *c*lass hierarchies, *m*ethods and *i*nheritance.

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 non-existent. 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.

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

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

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 of inheritance. The last statement is pondered by "in principle" since, to date, code reuse of real world objects is still far from being commonplace.

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 importantly, highly maintainable. Better than an argumentation, we show how to implement some numerical algorithms selected among those that, in my opinion, are most useful for the areas where object-oriented software is used primarily: finance, medicine and decision support.

1.2 Object-oriented concepts in a nutshell

First let us define what is covered by the adjective *o*bject-oriented. Many software vendors are qualifying a piece of software 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 *V*isual in their names[3]. Visual programming is something entirely different from object-oriented programming.

Object-oriented is 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.

*D*ata 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 *h*iding the implementation, a more precise description of what is

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

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 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 than 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 Smalltalk, one of the best object-oriented programming language. For this book, we are using Pharo[4], a modern open-source implementation of Smalltalk.

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.

Floating point representation

Currently mankind is using the decimal system[5]. In this system, however, most rational numbers and all irrational and transcendental numbers es-

[4] http://www.pharo.org/

[5] This is of course quite fortuitous. Some civilizations have opted for a different base. The Sumerians have used the base 60 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 book of Georges Ifrah [**?**].

cape our way of representation. Numbers such as 1/3 or π cannot be written in the decimal system other than approximately. One can chose 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 represented by a finite decimal representation. This kind of limitation has nothing to do with the use of computers. To go around that limitation, mathematicians have invented abstract representations of numbers, which can be manipulated in regular computations. This includes irreducible fractions (1/3 *e*.g.), irrational numbers ($\sqrt{2}$ *e*.g.), transcendental numbers (π and e the base of natural logarithms *e*.g.) and normal[6] 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 *a*d infinitum. Probably, such a computer would be much slower. Using digital representation — the word digital being taken in its first sense, that is, a representation with digits — no matter how clever the implementation[7], 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[8]: an integral part called the mantissa m and an exponent e. This way of doing is quite familiar to people using large quantities (astronomers *e*.g.) or studying the microscopic world (microbiologists *e*.g.). Of course, the natural radix for people is 10. For example, the average distance from earth to sun expressed in kilometer is written as 1.4959787×10^8.

In the case of radix 2, the number 18446744073709551616 is represented as 1×2^{64}. Quite a short hand compared to the decimal notation! IEEE standard floating-point numbers use 24 bits for the mantissa (about 8 decimal digits)

[6] Since Cantor, mathematicians have learned that there are many kinds of infinities. See for example reference [**?**].

[7] 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.

[8] This 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.

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.

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 of all, 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 errors have the non-trivial 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[9].

Other type 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 just off by 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 3.6×10^{-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 of what reference

[9] In modern days microprocessor, a floating point arithmetic unit actually uses more digits than the representation. These extra digits are called *g*uard digits. Such difference is not relevant for our example.

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 solutions to that equation are:

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

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, which 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 going out of the scope of this book. This section was only meant as a reminder that numerical results coming out from a computer must always be taken with a gain of salt. This 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 Knuth [**?**]. The excellent article by David Goldberg — *W*hat every computer scientist should know about floating point arithmetic, published in the March 1991 issues of Computing Surveys — is recommend for a quick, but in-depth, survey. This article can be obtained from various WEB sites. Let us conclude this section with a quotation from Donald E. Knuth [**?**].

> *F*loating 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.

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.

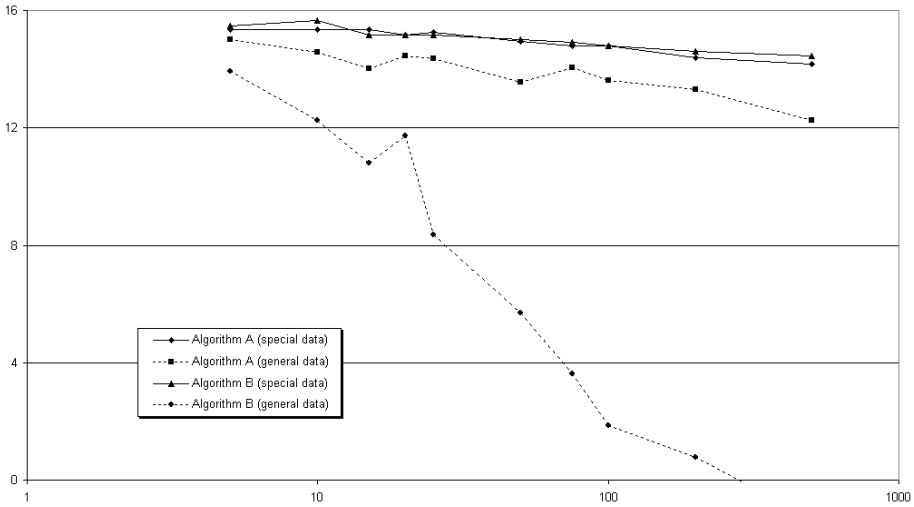


Figure 1-1 Comparison of achieved precision

This numerical problem has one parameter, which measures the complexity of the data. Moreover data can be of two types: general data or special data. Special data have some symmetry properties, which 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.

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 chose 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 above. Thus, for general data, one has no choice but to use algorithm

A.

Readers who do not like mysteries can go read section **??** where these algorithms are discussed.

Outsmarting rounding errors

In some instances rounding errors can be significantly reduced if one spends some time reconsidering how to compute the final solution. In this section we like to show an example of such thinking.

Consider the following second order equation, which must be solved when looking for the eigenvalues of a symmetric matrix (*c.f.* section **??**):

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

Without restricting the generality of the argumentation, we shall assume that α is positive. the problem is to find the 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:

$$t_{\text{min}} = \sqrt{\alpha^2 + 1} - \alpha. \tag{1.2}$$

Let us now assume that α is very large, so large that adding 1 to α^2 cannot be noticed within the machine precision. Then, the smallest of the solutions of equation 1.1 becomes $t_{\text{min}} \approx 0$, 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:

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

Now we have for large α:

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

This solution has certainly some rounding errors, but much less 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 zero for large α, as it should be.

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 a full control of 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 non-sensical precision.

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

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

$$\frac{p_{\mathrm{i}n}}{d} < \pi < \frac{p_{\mathrm{O}ut}}{d}. \tag{1.6}$$

By increasing n, one can improve the precision of the determination of π. During the Antiquity and the Middle Age, 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 π. In 1424, Jamshid Masud al-Kashi, a persian scientist, published an approximation of π with 16 decimal digits. The number of sides of the polygons was 3×2^8. This was quite an achievement, the last of its kind. After that, mathematicians discovered other means of expressing the number π.

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 π was such that,

> *t*he error in computing the perimeter of a circle with a radius 600000 times that of earth would be less than the thickness of a horse's hair.

The reader should know that the *t*hickness 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 Jamshid Masud al-Kashi is 147 times the distance between the sun and the earth, or about 3 times the radius of the orbit of Pluto, the most distant planet of the solar system.

As Jan Gullberg notes in his book, *a*l-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:

> *D*o 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 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 about the concept of *W*rite 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 *e*t 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 which are relevant for numerical computations. These parameters correspond to the instance variables of the object responsible to compute them. They are the following:

the radix of the floating-point representation, that is r.

the largest positive number which, when added to 1 yields 1.

the largest positive number which, when subtracted from 1 yields 1.

the smallest positive number different from 0.

the largest positive number which can be represented in the machine.

the relative precision, which can be expected for a general numerical computation.

a number, which 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 (*c*.f. next paragraph) assuming the radix is 2. Then, one keeps adding 1 to this number until the result changes. The number of added ones is the radix.

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

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

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 compute 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, which 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[10], 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.

Computer numerical precision

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.

Listing below shows the class `PMFloatingPointMachine` responsible of 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.

[10] Of course, after making sure that the ratio is well defined numerically.

```
Object subclass: #PMFloatingPointMachine
  instanceVariableNames: 'defaultNumericalPrecision radix
    machinePrecision negativeMachinePrecision smallestNumber
    largestNumber smallNumber largestExponentArgument'
  classVariableNames: 'UniqueInstance'
  package: 'Math-Core'
```

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

```
PMFloatingPointMachineMachine >> reset
   UniqueInstance := nil
```

The computation of the smallest and largest numbers uses exceptions to detect the underflow and the overflow.

```
PMFloatingPointMachine >> 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].
```

```
PMFloatingPointMachine >> 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].
```

```
PMFloatingPointMachine >> 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].
```

```
PMFloatingPointMachine >>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: [].
```

```
PMFloatingPointMachine >> 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 ].
```

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

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

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

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

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

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

The method `showParameters` can be used to print the values of the parameters onto the Transcript window.

```
PMFloatingPointMachine >> 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.
  Transcript flush
```

```
PMFloatingPointMachine >> smallestNumber
    smallestNumber isNil
        ifTrue: [ self computeSmallestNumber ].
    ^smallestNumber
```

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

1.5 Comparing floating point numbers

It is very surprising to see how frequently questions about the lack of equality between two floating-point numbers are posted on the Smalltalk electronic discussion groups. As we have seen in section 1.3 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.

> radix machine, [illegible] 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 ϵ is smaller than a given number $\epsilon_{\max}$. If the denominator of the fraction on equation 1.9 is less than $\epsilon_{\max}$, than the two numbers can be considered as being 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.

In Smalltalk this means adding a new method to the class Number as shown in Listing 1-2.

Listing 1-2 Comparison of floating point numbers in Smalltalk

```
Number >> equalsTo: aNumber
   ^self relativelyEqualsTo: aNumber upTo:
         PMFloatingPointMachine new defaultNumericalPrecision
```

```
Number >> relativelyEqualsTo: aNumber upTo: aSmallNumber
   | norm |
   norm := self abs max: aNumber abs.
   ^norm <= PMFloatingPointMachine new defaultNumericalPrecision
    or: [ (self - aNumber) abs < ( aSmallNumber * norm)]
```

1.6 Speed consideration (to be revisited)

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

First of all, 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 if the application is interactive. For example, performing a least square fit to a histogram in Smalltalk and drawing the resulting fitted function is usually hardly perceptible to the eye on a personal computer. 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 for its maintainability. As 80% of the code development is spent in maintenance this aspect should first be considered.

Table 1-3 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 [**?**] 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. Elapsed time were measured by repeating the measured evaluation a sufficient number of time so that the error caused by the CPU clock is less that the last digit shown in the final result.

Table 1-3 Compared execution speed between C and Smalltalk

Operation	**U**nits	**C**	**S**malltalk
Polynomial 10 degree	msec.	1.1	27.7
Neville interpolation (20 points)	msec.	0.9	11.0
LUP matrix inversion (100×100)	sec.	3.9	22.9

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.

I have successfully build data mining Smalltalk applications using **a**ll 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.

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. (*c.f.* section **??** for definitions of these terms).

There is, however, a price to pay for the perfection offered by fractions. When using fractions, the computing time often becomes prohibitive. Resulting fractions are often composed of large integers. This 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 301×301 matrix with the code of section **??**. The numbers used to build the matrix where obtained

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

from a measuring device (using an ADC) and where thus integers. The inversion time was over 2 hours[12]. 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[13] 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 can become prohibitive.

1.7 Conventions

Equations presented in this book are using standard international mathematical notation as described in [**?**]. 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:

Main 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, there are a few instances where we have departed from the widely used conventions.

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 [**?**].

Figure 1-4 shows a typical class diagram. A rectangular box with 2 or 3 parts represents a class. The top part contains the name of the class in bold face. If the class is an abstract class the name in shown in italic bold face. In figure 1-4 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 italic, for example `abstractMethod3` in the class `MyAbstractClass` of figure 1-4. The third part of the class box, if any,

[12] This particular customer was a very patient person!

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

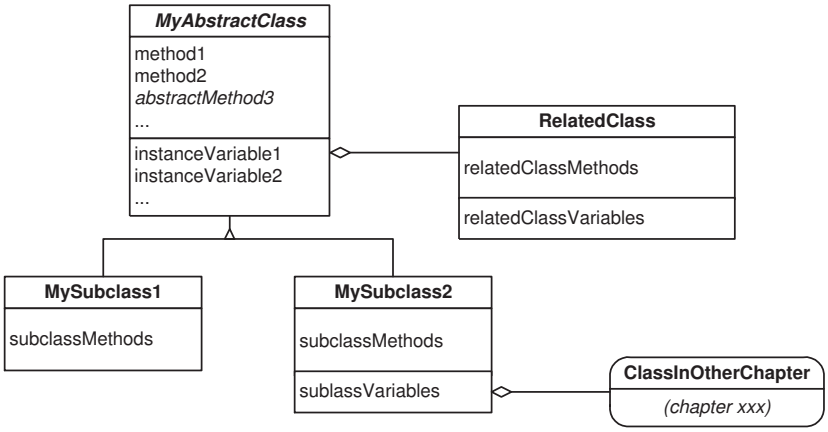


Figure 1-4 A typical class diagram

contains the list of all instance variables. If the class does not have any instance variable the class box only consists of 2 parts, for example the class `MySubClass1` of figure 1-4.

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 1-4. A horizontal line beginning with a diamond (UML aggregation symbol) indicates the class of an instance variable. For example, Figure 1-4 indicates that the instance variable `instanceVariable1` of the class `MyAbstractClass` must be an instance of the class `RelatedClass`. The diamond is black is the instance variable is a collection of instances of the class. A class within a rectangle with rounded corner represents a class already discussed in an earlier chapter; the reference to the chapter is written below the class name. Class `ClassInOtherChapter` in figure 1-4 is such a class. To save space, we have used the Smalltalk method names. It is quite easy to identify methods needing parameters when one uses Smalltalk method names: a colon in the middle or at the end of the method name indicates a parameter. Please refer to appendix **??** for more details on Smalltalk methods.

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 3-letter code identifying the origin of the code. In this book the names of the Smalltalk classes are all prefixed with PM like PolyMath[14].

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

[14]Classes were previously prefixed with author's initials.

to implement an initialization method, which 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 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 (*c.f.* section 1.4). Finally constants which are not likely to change can be stored in the code. This is acceptable as long as this 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 [**?**].

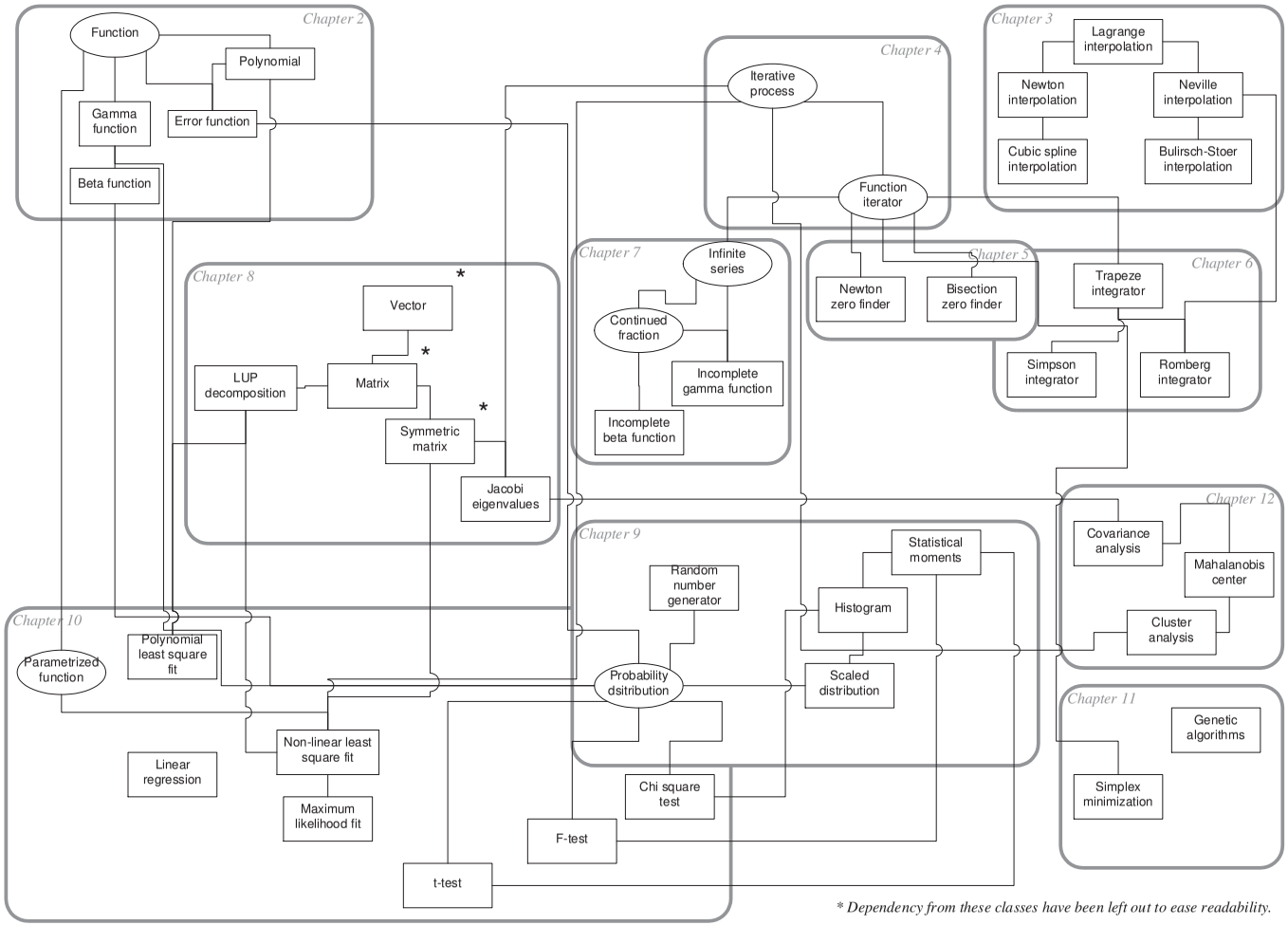
In [**?**] it is recommended to use the method name `default` to implement a singleton class. In this book this convention is not followed. In Smalltalk, however, the normal instance creation method is `new`. Introducing a method `default` for singleton classes has the effect of departing from this more ancient convention. In fact, requiring the use of `default` amounts to reveal 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.

1.8 Roadmap

This last section of the introduction describes the roadmap of the algorithms discussed in the book chapter by chapter. Figure 1-5 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 where the classes are discussed are drawn as grayed rectangles with rounded corners. Hopefully 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 1-5 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. Fi-

Figure 1-5 Book roadmap



nally three library functions are given: the error function, the gamma function and the beta function.

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

Chapter **??** 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.

Chapter **??** 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 **??**.

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

Chapter **??** discusses the specialization of the general framework of chapter **??** 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 part of this book.

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

Chapter **??** discussed the test of hypothesis and estimation. It gives 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 chapter 2, **??**, **??** 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 are using the general framework for iterative process discussed in chapter **??**.

Chapter **??** discusses the modern data mining techniques: correlation analysis, cluster analysis and neural networks. A couple of methods invented by the author 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, by a recurrence formula or by 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[2]. 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 [**?**] 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 on 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 imple-

[1] The absence of answer does not justify the absence of question.
[2] The good news is that they will if they read the present book (*c.f.* chapter **??**).

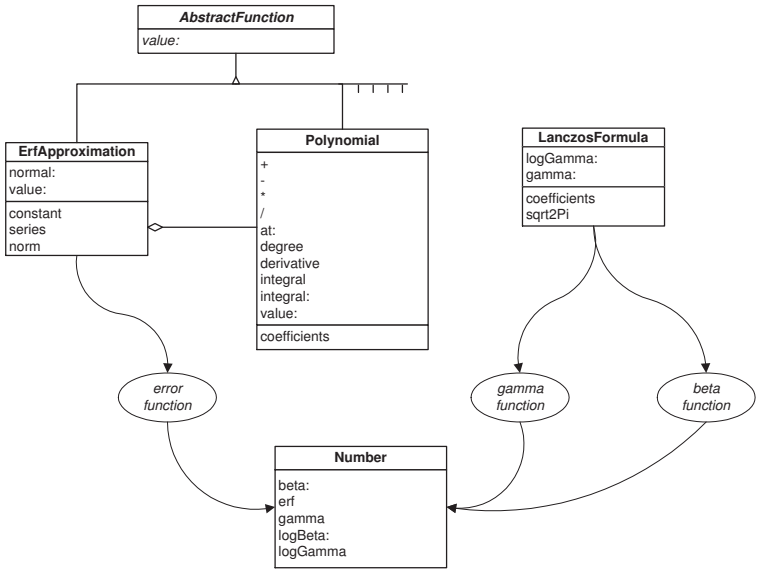


Figure 2-1 Smalltalk classes related to functions

mented as library functions — such as a sine, log or exponential — instead of using the general function formalism described in the first section.

Figure 2-1 shows the diagram of the Smalltalk classes described in this chapter. Here we have used special notations to indicate that the functions are implemented as library functions. The functions are represented by oval and arrows shows which class is used to implement a function for the class `Number`.

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 multi-variable function. Other types of variables are possible but will not be covered in this book.

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

2.2 Function – Smalltalk implementation

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 vari-

able is an array of values one talks about a multi-variable function. Other types of variables are possible but will not be covered in this book.

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

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].
```

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

```
f value: 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. In order 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 multi-variable function, the argument of the method `value:` is an Array or a vector (*c.f.* section **??**). Thus, in Smalltalk multi-variable functions can follow the same polymorphism as for one-variable functions.

Figure 2-1 with the box `AbstractFu` grayed.

2.3 Polynomials

Polynomials are quite important in numerical methods because they are often used in approximating functions. For example, section 2.4 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 (*c.f.* section **??**). Using this polynomial the correction for a given temperature can be evaluated for any temperature within the measured range.

The reader is advised to read carefully the implementation section as many techniques are introduced at this occasion. Later on those techniques will be mentioned with no further explanations.

Mathematical definitions

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

$$P(x) = \sum_{k=0}^{n} a_k x^k. \tag{2.2}$$

n is called the degree of the polynomial. For example, the second order polynomial

$$x^2 - 3x + 2 \tag{2.3}$$

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 2.2 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 2.3 is represented by the array $\{2, -3, 1\}$.

Evaluating equation 2.2 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]. The following formula shows the resulting expression:

Main equation⇒

$$(x) = a_0 + x\{a_1 + x[a_2 + x(a_3 + \cdots)]\} \tag{2.4}$$

Evaluating the above expression now requires only multiplications. The resulting algorithm is quite straightforward to implement. Expression 2.4 is called Horner's rule because it was first published by W.G. Horner in 1819. 150 years earlier, however, Isaac Newton was already using this method to evaluate polynomials.

In section **??** we shall requires the derivative of a function. For polynomials this is rather straightforward. The derivative is given by:

$$\frac{dP(x)}{dx} = \sum_{k=1}^{n} k a_k x^{k-1}. \tag{2.5}$$

[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.

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:

$$a'_k = (k+1)\, a_{k+1} \quad \text{for } k = 0, \ldots, n-1. \tag{2.6}$$

For example, the derivative of 2.3 is $2x - 3$.

The integral of a polynomial is given by:

$$\int_0^x P(t)\, dt = \sum_{k=0}^{n} \frac{a_k}{k+1} x^{k+1}. \tag{2.7}$$

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:

$$\bar{a}_k = \frac{a_{k-1}}{k} \quad \text{for } k = 1, \ldots, n+1. \tag{2.8}$$

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 2.3 which has the value -2 at $x = 0$ is the polynomial

$$\frac{x^3}{3} - \frac{3^2}{2} + 2x - 2. \tag{2.9}$$

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

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:

$$c_k = \sum_{i+j=k} a_i b_j \quad \text{for } k = 0, \ldots, n+m. \tag{2.10}$$

In equation 2.10 the coefficients a_k are treated as 0 if $k > n$. Similarly the coefficients n_k are treated as 0 if $k > m$.

Dividing a polynomial by another is akin to integer division with remainder. In other word the following equation:

$$P(x) = Q(x) \cdot T(x) + R(x). \tag{2.11}$$

[4] Notice the change in the range of the summation index in equation 2.5.
[5] The set of polynomials is a vector space in addition to being a ring.

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 [**?**].

Polynomial — Smalltalk 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 methods implements equation 2.4.

The methods `derivative` and `integral` return each a new instance of a polynomial. The method `integral:` must have an argument specifying the value of the integral of the polynomial at 0. A convenience `integral` method without a<rgument is equivalent to call 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 appendix (*c.f.* section **??**). The need for double dispatching comes for allowing an operation between object of different nature. 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.

Figure 2-1 with the box **P**olynomial grayed.

Being a special case of a function a polynomial must of course implement the behavior of functions as discussed in section 2.2. Here is a code example on how to use the class `PMPolynomial`.

Listing 2-2 How to use PMPolynomial

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

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

The next example shows how to manipulate polynomials in symbolic form.

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

```
polD := polynomial derivative.
polI := polynomial integral.
```

The first line creates the polynomial of example 2.3. The second line creates the polynomial $x^3 + 2x^2 + 7x - 3$. The third line of the code creates a new polynomial, product of the first two. The last two lines create two polynomials, respectively the derivative and the integral of the polynomial created in the third line.

Listing **??** shows the Smalltalk implementation of the class `PMPolynomial`.

A beginner may have been tempted to make `PMPolynomial` 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 `PMPolynomial` a subclass of Object. It has a single instance variable, an 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, in particular 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 if two different variables are pointing to the same object. Implementing a method `printOn:` allows displaying parameters particular to each instance so that the instances can easily be identified. It may also be used in quick print on the `Transcript` and may save you the use on an inspector while debugging. Implementing a method `printOn:` for each class that you create is a good general practice, which can make your life as a Smalltalker much easier.

Working with indices in Smalltalk is somewhat awkward for mathematical formulas because 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:` (*c.f.* 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 of the coefficients is done in the creation method to make things transparent.

The methods `derivative` and `integral` return a new instance of the class `PMPolynomial`. 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 (*c.f.* 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 0. In addition this method returns zero for any index larger than the polynomial's degree. This allows being lax with the index range. In particular, equation 2.10 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 dynamical 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, our implementation of polynomial is also valid for polynomials with complex coefficients.

Listing 2-4 Smalltalk implementation of the polynomial class

```
Object subclass: #PMPolynomial
   instanceVariableNames: 'coefficients'
   classVariableNames: ''
   package: 'Math-Polynomials'
```

```
PMPolynomial >> * aNumberOrPolynomial
   ^aNumberOrPolynomial timesPolynomial: self
```

```
PMPolynomial >> + aNumberOrPolynomial
    ^aNumberOrPolynomial addPolynomial: self
```

```
PMPolynomial >> - aNumberOrPolynomial
    ^aNumberOrPolynomial subtractToPolynomial: self
```

```
PMPolynomial >> / aNumberOrPolynomial
    ^aNumberOrPolynomial dividingPolynomial: self
```

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

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

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

```
PMPolynomial >> coefficients
    ^coefficients deepCopy
```

```
PMPolynomial >> degree
    ^coefficients size - 1
```

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

```
PMPolynomial >> dividingPolynomial: aPolynomial
    ^ (self dividingPolynomialWithRemainder: aPolynomial) first
```

```
PMPolynomial >> 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))
```

```
PMPolynomial >> initialize: anArray
    coefficients := anArray.
    ^ self
```

```
PMPolynomial >> integral
    ^ self integral: 0
```

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

```
PMPolynomial >> 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.
        ]
```

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

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

```
Polynomial >> 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
```

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

Listing 2-5 shows the listing of the methods used by the class Number as part of the double dispatching of the arithmetic operations on polynomials.

Listing 2-5 Methods of class Number related to polynomials

```
Number >> beta: aNumber
    ^ (self logBeta: aNumber) exp
```

```
Number >> logBeta: aNumber
    ^ self logGamma + aNumber logGamma - ( self + aNumber) logGamma
```

2.4 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 2.4 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 in percent. For example, obstetricians look whether the weight at birth of the first born child is located below the 10^{th} centile or above the 90^{th} centile to assess a risk factor for a second pregnancy[6].

[6] c.f. footnote 8 on page 39

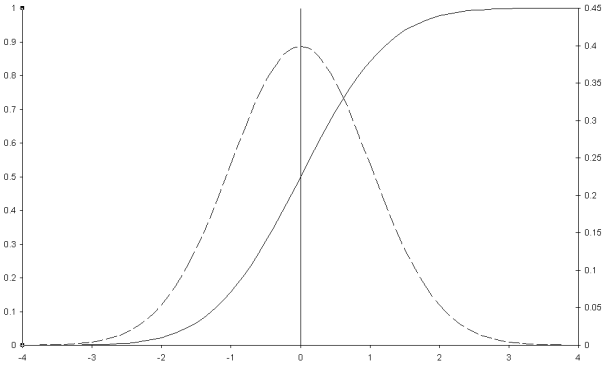


Figure 2-6 The error function and the normal distribution

Mathematical definitions

Because it is the integral of the normal distribution, the error function, $erf\,(x)$, gives the probability of finding a value lower than x when the values are distributed according to a normal distribution with mean 0 and standard deviation 1. The mean and the standard deviation are explained in section **??**. This probability is expressed by the following integral[7]:

$$erf\,(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{t^2}{2}} dt \tag{2.12}$$

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

One could carry out the integral numerically, but there exists several good approximations. The following formula is taken from [**?**].

Main equation⇒

$$erf\,(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} \sum_{i=1}^{5} a_i r\,(x)^i \quad \text{for } x \geq 0. \tag{2.13}$$

where

$$r\,(x) = \frac{1}{1 - 0.2316419x}. \tag{2.14}$$

and

$$\begin{cases} a_1 = & 0.31938153 \\ a_2 = & -0.356563782 \\ a_3 = & 1.7814779372 \\ a_4 = & -1.821255978 \\ a_5 = & 1.330274429 \end{cases} \tag{2.15}$$

[7] In [**?**] and [**?**], the error function is defined as:

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

.

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

$$erf(x) = 1 - erf(-x). \tag{2.16}$$

⇐Main equation

When dealing with a general Gaussian distribution with average μ and standard deviation σ it is convenient to define a generalized error function as:

$$erf(x; \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} \int_{-\infty}^{x} e^{-\frac{(x-\mu)^2}{2\sigma^2}} dt. \tag{2.17}$$

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

$$erf(x; \mu, \sigma) = erf\left(\frac{x-\mu}{\sigma}\right). \tag{2.18}$$

⇐Main equation

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 a Gaussian distribution with average μ and standard deviation σ:

$$Prob\left(\frac{|x-\mu|}{\sigma} \leq t\right) = 2 \cdot erf(t) - 1. \quad \text{for } t \geq 0. \tag{2.19}$$

Example

Now we can give the answer to 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 $2.85\,\mathrm{K}g$. 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 $3.39\,\mathrm{K}g$ and a standard deviation of $0.44\,\mathrm{K}g$[8]. The probability of having a weight of birth smaller than that of the woman's first child is:

$$\begin{aligned} Prob(Weight \leq 2.85\,\mathrm{K}g) &= erf\left(\frac{2.85 - 3.39}{0.44}\right), \\ &= 11.2\%. \end{aligned}$$

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

Error function — Smalltalk implementation

The error function is implemented as a single method for the class Number. Thus, computing the centile of our preceding example is simply coded as:

Figure 2-1 with the box ErfApproxi grayed.

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

```
| 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 lay within 3 standard deviations from its mean, you need to evaluate the following expression using equation 2.19:

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

If one needs to use the error function as a function, one must use it inside a block closure. In this case one defines a function object as follows:

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

Listing 2-7 shows the Smalltalk implementation of the error function.

In Smalltalk we are allowed 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, `PMErfApproximation`, used to store the constants of equation 2.15 and evaluate the formula of equations 2.13 and 2.14. In our case, there is no need to create a separate instance of the class `PMErfApproximation` at each time since all instances would actually be exactly identical. Thus, the class `PMErfApproximation` is a singleton class. A singleton class is a class, which can only create a single instance [**?**]. 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 [**?**].

Listing 2-7 Smalltalk implementation of the Error function

```
Number >> errorFunction
    ^PMErfApproximation new value: self
```

```
Object subclass: #PMErfApproximation
   instanceVariableNames: 'constant series norm'
   classVariableNames: 'UniqueInstance'
   package: 'Math-Core-Distribution'
```

```
PMErfApproximation class >> new
    UniqueInstance isNil
        ifTrue: [UniqueInstance := super new initialize].
    ^UniqueInstance
```

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

```
PMErfApproximation >> normal: aNumber
   "Computes the value of the Normal distribution for aNumber"

   ^ [ (aNumber squared * -0.5) exp * norm ]
     on: Error
     do: [ :signal | signal return: 0 ]
```

```
PMErfApproximation >> 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)
```

2.5 Gamma function

The gamma function is used in many mathematical functions. In this book, the gamma function is needed to compute the normalization factor of several probability density functions (*c.f.* sections **??** and **??**). It is also needed to compute the beta function (*c.f.* section 2.6).

Mathematical definitions

The Gamma function is defined by the following integral, called Euler's integral[9]:

$$\Gamma(x) = \int_0^\infty t^x e^{-t} dt \tag{2.20}$$

From equation 2.20 a recurrence formula can be derived:

$$\Gamma(x+1) = x \cdot \Gamma(x) \tag{2.21}$$

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

$$\begin{cases} \Gamma(1) = 1 \\ \Gamma(2) = 1 \end{cases} \tag{2.22}$$

[9] Leonard Euler to be precise as the Euler family produced many mathematicians.

From 2.21 and 2.22, the well-known relation between the value of the Gamma function for positive integers and the factorial can be derived:

$$\Gamma(n) = (n-1)! \quad \text{for } n > 0. \tag{2.23}$$

The most precise approximation for the Gamma function is given by a formula discovered by Lanczos [**?**]:

Main equation⇒

$$\Gamma(x) \approx e^{\left(x+\frac{5}{2}\right)} \left(x+\frac{5}{2}\right) \frac{\sqrt{2\pi}}{x} \left(c_0 + \sum_{n=1}^{6} \frac{c_n}{x+n} + \epsilon\right) \tag{2.24}$$

where

$$\begin{cases} c_0 = & 1.000000000190015 & \\ c_1 = & 76.18009172947146 & \\ c_2 = & -86.50532032941677 & \\ c_3 = & 24.01409824083091 & \\ c_4 = & -1.231739572450155 & \\ c_5 = & 1.208650973866179 & \cdot 10^{-3} \\ c_6 = & -5.395239384953 & \cdot 10^{-6} \end{cases} \tag{2.25}$$

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' formula with the recurrence formula 2.21 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 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 is seldom used, the value of the Gamma function for negative non-integer numbers can be computed using the reflection formula hereafter:

$$\Gamma(x) = \frac{\pi}{\Gamma(1-x)\sin \pi x} \tag{2.26}$$

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

1. If x is a non-positive integer ($x \leq 0$), raise an exception.
2. If x is smaller than or equal to 1 ($x < 1$), use the recurrence formula 2.21.
3. If x is negative ($x < 0$, but non integer), use the reflection formula 2.26.
4. Otherwise use Lanczos' formula 2.24.

One can see from the leading term of Lanczos' formula that the gamma function raises faster than an exponential. Thus, evaluating the gamma function for numbers larger than a few hundreds 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 following expression:

$$\frac{\Gamma(460.5)}{\Gamma(456.3)} \tag{2.27}$$

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 2.27 is then evaluated as:

$$\exp\left[\ln\Gamma(460.5) - \ln\Gamma(456.3)\right] \tag{2.28}$$

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

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

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:

```
2.5 gamma
```

To obtain the logarithm of the gamma function, you need to evaluate the following expression:

```
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 2.4 for detailed explanations.

Since the method factorial is already defined for integers in the base classes, the gamma function has been defined using equation 2.23 for integers. An error is generated if one attempts to compute the gamma function for non-positive integers. The class Number delegates the computation of Lanczos' formula to a singleton class. This is used by the non-integer subclasses of Number: Float and Fraction.

The execution time to compute the gamma function for floating argument given in Table 1-3 in section 1.6.

Figure 2-1 with the box LanczosFor grayed.

Listing 2-8 Smalltalk implementation of the gamma function

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

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

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

```
Object subclass: #PMLanczosFormula
   instanceVariableNames: 'coefficients sqrt2Pi'
   classVariableNames: 'UniqueInstance'
   package: 'Math-DHB-Numerical'
```

```
PMLanczosForumula class >> new
   UniqueInstance isNil
      ifTrue: [ UniqueInstance := super new initialize ].
   ^ UniqueInstance
```

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

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

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

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

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

2.6 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 (c.f. sections **??**, **??** and **??**).

Mathematical definitions

The beta function is defined by the following integral:

$$B(x,y) = \int_0^1 t^{x-1}(1-t)^{y-1}\,dt \tag{2.29}$$

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

$$B(x,y) = \frac{\Gamma(x)\,\Gamma(y)}{\Gamma(x+y)} \tag{2.30}$$

Thus, computation of the beta function is directly obtained from the gamma function. As evaluating the gamma function might overflow the floating-point exponent (c.f. discussion at the end of section 2.5), it is best to evaluate the above formula using the logarithm of the gamma function.

Beta function — Smalltalk implementation

Like the error and gamma functions, the gamma 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:

Listing 2-9 2

```
.5 beta: 5.5
```

Computing the logarithm of the beta function of 2.5 and 5.5 is simply coded as:

Listing 2-10 2

```
.5 logBeta: 5.5
```

Listing **??** shows the implementation of the beta function in Smalltalk.

Figure 2-1 with the box LanczosFor grayed.

Listing 2-11 Smalltalk implementation of the beta function

```
\input{Smalltalk/FunctionEvaluation/Number(DhbBetaFunction).tex}
```

CHAPTER 3

Technical requirements

3.1 Motivations

Reasons (fonts, packages)

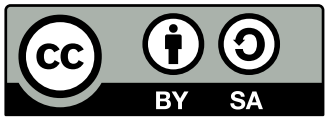
3.2 A pretty up-to-date TeXlive

how to update important packages

3.3 Building with LuaLaTeX

how to run, latexmk

Figure 3-1 A rather large representation of the icon for the Creative Commons license we at Square Bracket Associates use for our books.



CHAPTER 4

Page layout and design

4.1 Font choices

In technical writing, and especially using LaTeX, it's tempting to try to use fonts as a semantic markup of sorts; unfortunately, this only results in a jumble of too many fonts and uselessly noisy paragraphs. Granted, a technical book can never be as minimal as a novel typeset in a single size, single weight roman font, but some restraint should help maintain a clear visual hierarchy. We've thus tried to keep to a minimal set of fonts with clearly defined roles. Thankfully, there has been several impressive open-source fonts released in the last years, which made several combinations possible; in the end, we settled on three families in Table 4-1, which are all packaged in the TeXlive distribution.

The workhorse font is Gentium[1]; it's compact, very legible, and not as cold as the fonts we usually see in LaTeX documents. In titles, Open Sans[2] gives a neutral but friendly impression; it is not overpowering and is very legible in small sizes, making it perfect for captions and page decorations. Finally, the font for source code excerpts and verbatim text is Fira Mono[3].

Table 4-1 Fonts used in the document, and their roles

Gentium – *Italic*	Primary, paragraph text
Open Sans – **Bold**	Structural and secondary text: titles, captions
`Fira Mono`	Verbatim text and code

[1] http://software.sil.org/gentium/ — SIL Open Font License
[2] https://www.google.com/fonts/specimen/Open+Sans — Apache License
[3] https://mozilla.github.io/Fira/ — SIL Open Font License

Table 4-2 Some convenient delimiters for inline code

`\code|...|`	pipe	`\code§...§`	section sign
`\code_..._`	underscore	`\code¶...¶`	pilcrow
`\code!...!`	exclamation point	`\code$...$`	dollar sign
`\code¡...¡`	inverse exclamation point	`\code~...~`	tilde
`\code?...?`	question mark	`\code^...^`	circumflex
`\code¿...¿`	inverse question mark	`\code`...``	backquote

4.2 Text layout

ragged right spaced paragraphs with no indent hanging section numbers

4.3 Custom semantic markup

chapterprecis constraints on content

4.4 Source code and listings

SBAbook provides custom high-level semantic markup for source code, which relies on the powerful `listings` and `tcolorbox` packages.

Inline source code For short mentions of source code in the middle of the text, use the `\code|...|` macro. This macro is an alias to `\lstinline`, which works like verbatim: it uses an arbitrary delimiter.

Using this macro with curly braces like `\code{this}` is possible, but this is an experimental feature of `listings`, and it breaks in some cases; it's fine when used in paragraph text, but things get weird as soon as it's part of the argument of some other macro. Table 4-2 lists a few characters in the ASCII range that are convenient as delimiters.

Displayed source code For multi-line excerpts of source code that should appear in the flow of the text, use the `displaycode` environment. You have to specify the language each excerpt of code is written in, as an argument; any language known to the `listings` package works.

```
10 PRINT "Hello SBAbook!"
20 PRINT "This is a paragraph-level listing."
25 PRINT "UTF-8 in Basic: élèves français, Ελλάδα, Здравствуй, мир!"
30 GOTO 10
```

Referenced listings If you need a numbered source code listing that is referenced in the table of contents, use the `listing` environment, providing a descriptive caption as a second mandatory argument. To reference that

listing from the text, the label must be provided as `label=something` in the optional argument, as in listing 4-3 below. Similarly, use the `list text` option from `tcolorbox` if the caption should be worded differently in the table of contents.

Listing 4-3 A factory method in class `Foo`

```
Foo class>>with: parameter
  ^ self new
    initializeWith: parameter
```

Floating listings To make a floating listing, simply add the `float` option in the environment's optional parameter. In case you really need to specify the placement, the usual specifiers can also be passed: `float=htbp`.

4.5 Packages and conventions