PolyMath

Serge Stinckwich, Stéphane Ducasse

October 1, 2023
commit 1c0158f*

[&] 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 to be

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

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 usually

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

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.

*C*lass 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.

*P*olymorphism 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 escape

[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 [**?**].

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 *guard 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.

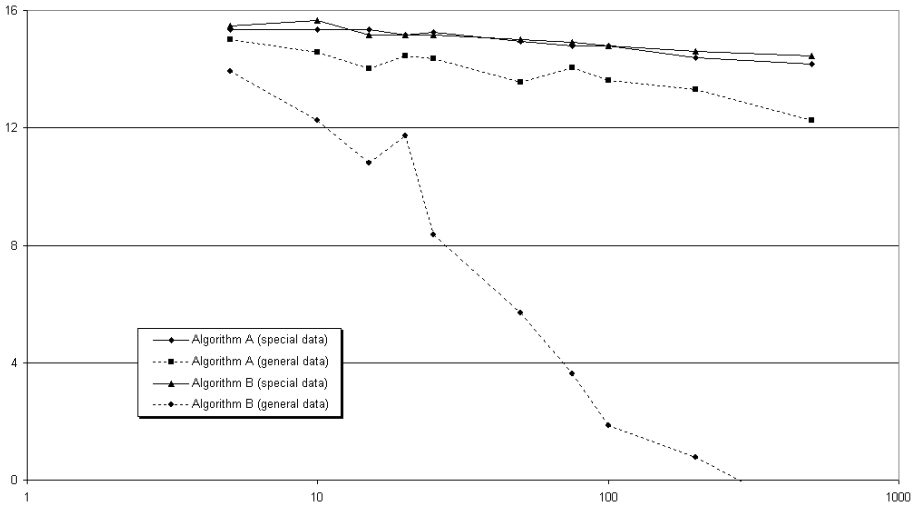


Figure 1-1 Comparison of achieved precision

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 smeasuring 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 8.5 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 8.7):

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

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_{in} of a n-sided regular polygon inscribed within the circle and the perimeter p_{out} of a n-sided regular polygon whose sides the tangent to the same circle. We have:

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

By increasing n, one can improve the precision of the determination of 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 *et al.* [**?**] shows a clever code determining all the parameters of the floating-point representation of a particular computer. In this book we shall concentrate only on the parameters 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
  | one floatingRadix fullMantissaNumber |
  one := 1.0.
  floatingRadix := self radix asFloat.
  fullMantissaNumber := one - ( floatingRadix * self
    negativeMachinePrecision).
  largestNumber := fullMantissaNumber.
  [[ fullMantissaNumber := fullMantissaNumber * floatingRadix.
      fullMantissaNumber isInfinite ifTrue: [^nil].
      largestNumber := fullMantissaNumber.
      true] whileTrue: [ ].
  ] on: Error do: [ :signal | signal return: 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.

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

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

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

The two numbers can be considered equal if 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	Units	C	Smalltalk
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 8.3 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 8.3. 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, contains the list of all instance variables. If the class does not have any instance

[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`

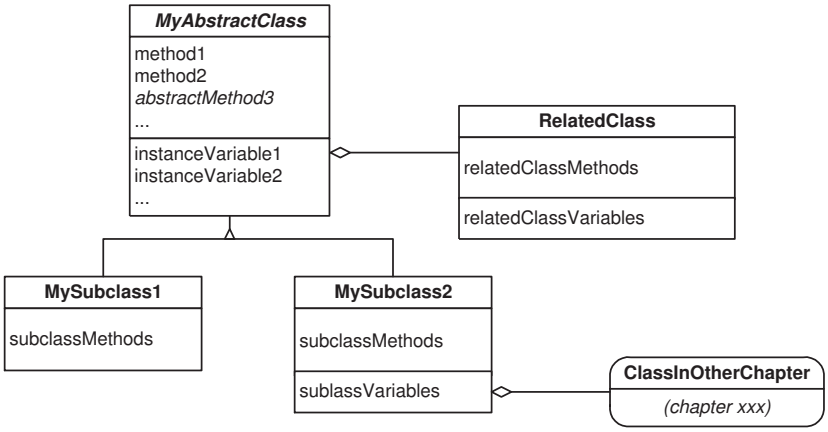


Figure 1-4 A typical class diagram

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 to implement an initialization method, which sets the desired values into the

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

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 [**?**].

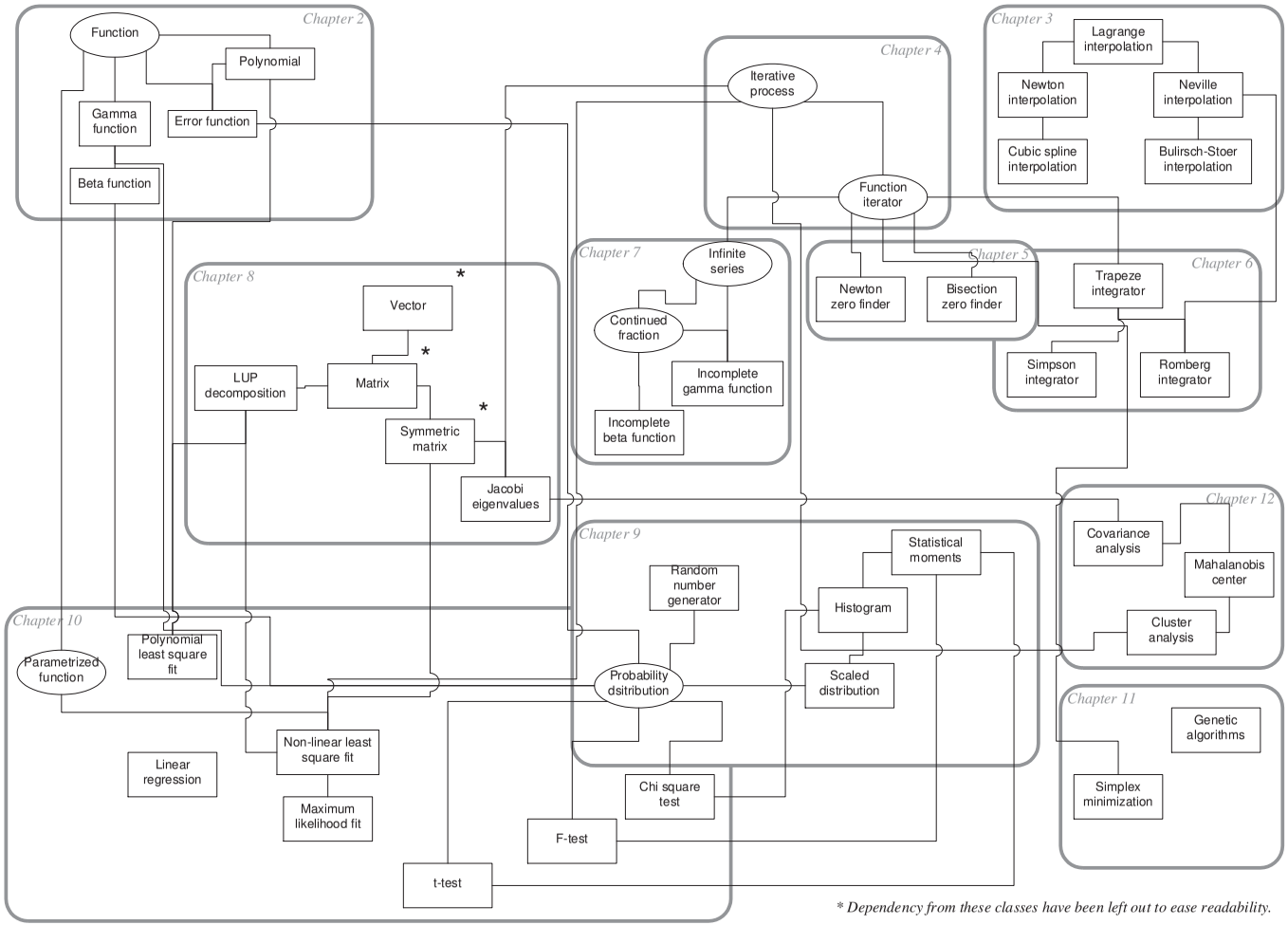
In [**?**] it is recommended to use the method name `default` to implement a singleton class. In this book this convention is not followed. In Smalltalk, however, the normal instance creation method is `new`. Introducing a method `default` for singleton classes has the effect of departing from this more 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 8 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.

Figure 1-5 Book roadmap



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

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

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

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

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

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

Chapter 8 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 9 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, 5 and 6.

Chapter 10 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, 4, 8 and 9.

Chapter 11 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 4.

Chapter 12 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 9 and 10. 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 6).

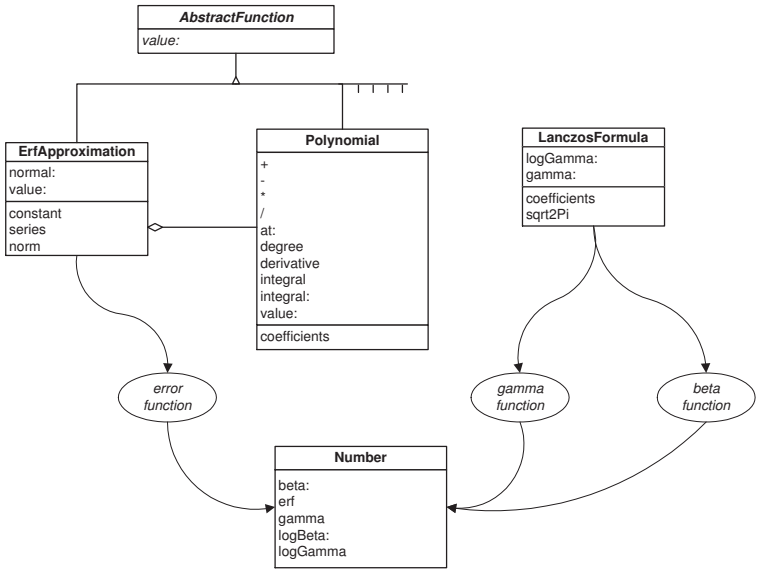


Figure 2-1 Smalltalk classes related to functions

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 8.1). Thus, in Smalltalk multi-variable functions can follow the same polymorphism as for one-variable functions.

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 10.8). 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:

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

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 5.3 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:

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

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 [**?**].

[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.

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.

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 polynomi-

als, 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)])
```

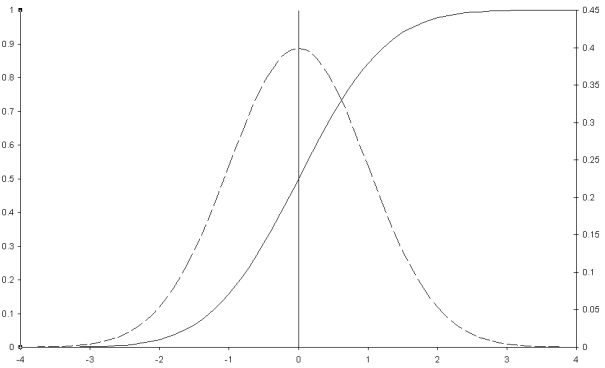


Figure 2-6 The error function and the normal distribution

```
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].

Mathematical definitions

Because it is the integral of the normal distribution, the error function, $\text{erf}\,(x)$, 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 9.1. This probability is expressed by the following integral[7]:

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

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 [**?**].

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

where

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

and

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

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

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

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

.

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:

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

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

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

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

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

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:

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

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 Kg. and let the duration of her first pregnancy be 39 weeks. In this case measurements over a representative sample of all births yielding healthy babies have an average of 3.39 Kg and a standard deviation of 0.44 Kg[8]. The probability of having a weight of birth smaller than that of the woman's first child is:

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

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

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:

[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 9.7 and 10.3). 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 [**?**]:

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

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.

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 10.1, 10.2 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:

```
2.5 beta: 5.5
```

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

```
2.5 logBeta: 5.5
```

Listing 2-9 shows the implementation of the beta function in Smalltalk.

Listing 2-9 Smalltalk implementation of the beta function

```
Number >> beta: aNumber
    "Computes the beta function of the receiver and aNumber"

    ^ (self logBeta: aNumber) exp
```

```
Number >> logBeta: aNumber
    "Computes the logarithm of the beta function of the receiver and
     aNumber"

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

CHAPTER 3

Interpolation

*O*n ne peut prévoir les choses qu'après qu'elles sont arrivées.[1]
Eugène Ionesco

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

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

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

Interpolation should not be mistaken with function (or curve) fitting. In the case of interpolation the sample points purely determine the interpolated function. Function fitting allows constraining the fitted function independently from the sample points. As a result fitted functions are more stable

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

than interpolated functions especially when the supplied values are subject to fluctuations coming from rounding or measurement errors. Fitting is discussed in chapter 10.

3.1 General remarks

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

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

In this chapter we discuss 3 interpolation functions: Lagrange interpolation polynomial, a diagonal rational function (Bulirsch-Stoer interpolation) and cubic spline. Furhermore, we show 3 different implementation of the Lagrange interpolation polynomial: direct implementation of Lagrange's formula, Newton's algorithm and Neville's algorithm. Figure 3.1 shows how the classes corresponding to the different interpolation methods described in this chapter are related to each other.

Definition

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

Examples

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

Figure 3.1 shows sample points — indicated by crosses — representing correction to the energy measured within a gamma ray detector made of several densely packed crystals. The energy is plotted on a logarithmic scale.

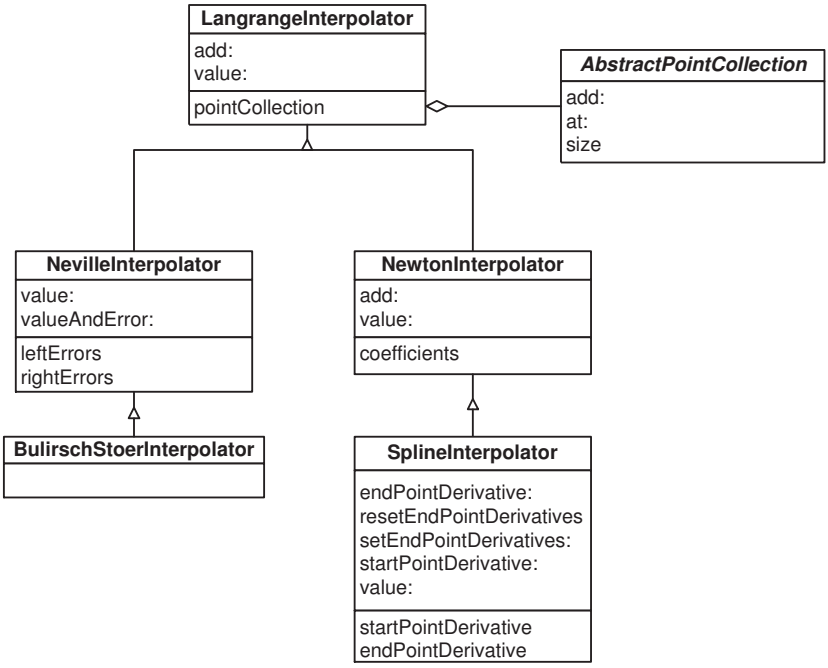


Figure 3-1 Class diagram for the interpolation classes

The correction is caused by the absorption of energy in the wrapping of each crystal. The sample points were computed using a simulation program[2], each point requiring several hours of computing time. Interpolation over these points was therefore used to allow a quick computation of the correction at any energy. This is the main point of this example: the determination of each point was expensive in terms of computing time, but the function represented by these points is continuous enough to be interpolated. The simulation program yields results with good precision so that the resulting data are not subjected to fluctuation.

The gray thick line in figure 3.1 shows the Lagrange interpolation polynomial obtained from the sample points. It readily shows limitations inherent to the use of interpolation polynomials. The reader can see that for values above 6.5 — corresponding to an energy of 500 MeV — the interpolated function does not reproduce the curve corresponding to the sample points. In fact, above 4.0 — that is, 50 MeV on the scale of figure 3.1 — the correction is expected to be a linear function of the logarithm of the energy.

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

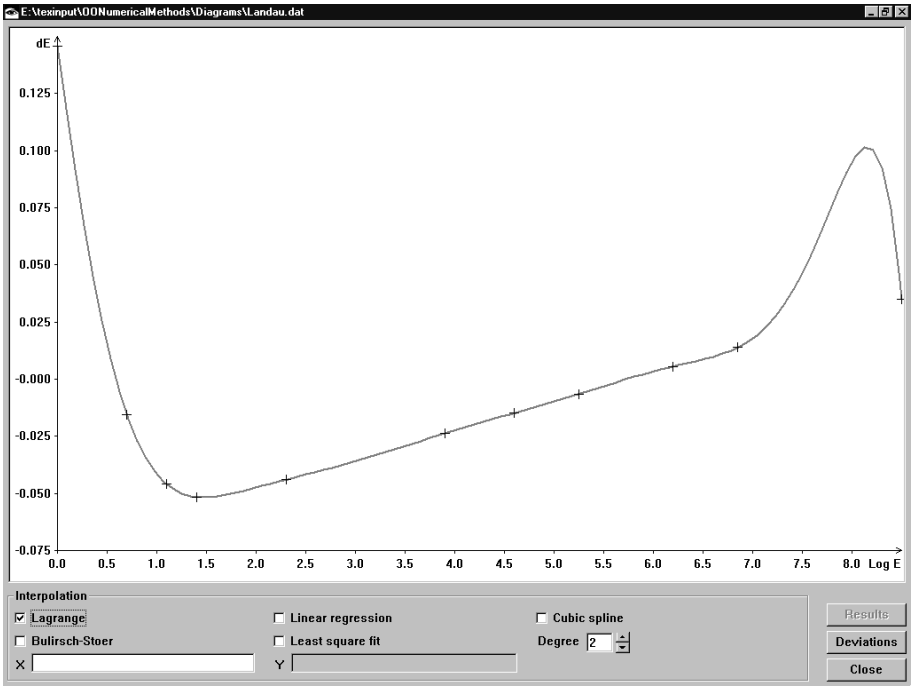


Figure 3-2 Example of interpolation with the Lagrange interpolation polynomial

Figure 3.1 shows a comparison between the Lagrange interpolation polynomial (gray thick line) and interpolation with a rational function (black dotted line) using the same sample points as in figure 3.1. The reader can see that, in the high-energy region (above 4 on the scale of figure 3.1) the rational function does a better job than the Lagrange polynomial. Between the first two points, however, the rational function fails to reproduce the expected behavior.

Figure 3.1 shows a comparison between the Lagrange interpolation polynomial (gray thick line) and cubic spline interpolation (black dotted line) using the same sample points as in figure 3.1. The reader can see that, in the high-energy region (above 4 on the scale of figure 3.1) the cubic spline does a better job than the Lagrange polynomial. In fact, since the dependence is linear over that range, the cubic spline reproduces the theoretical dependence exactly. In the low energy region, however, cubic spline interpolation fails to reproduce the curvature of the theoretical function because of the limitation of the polynomial's degree.

A final example shows a case where interpolation should not be used. Here the sample points represent the dependence of the probability that a coin mechanism accepts a wrong coin as a function of an adjustable threshold. The determination of each point requires 5-10 minutes of computing time.

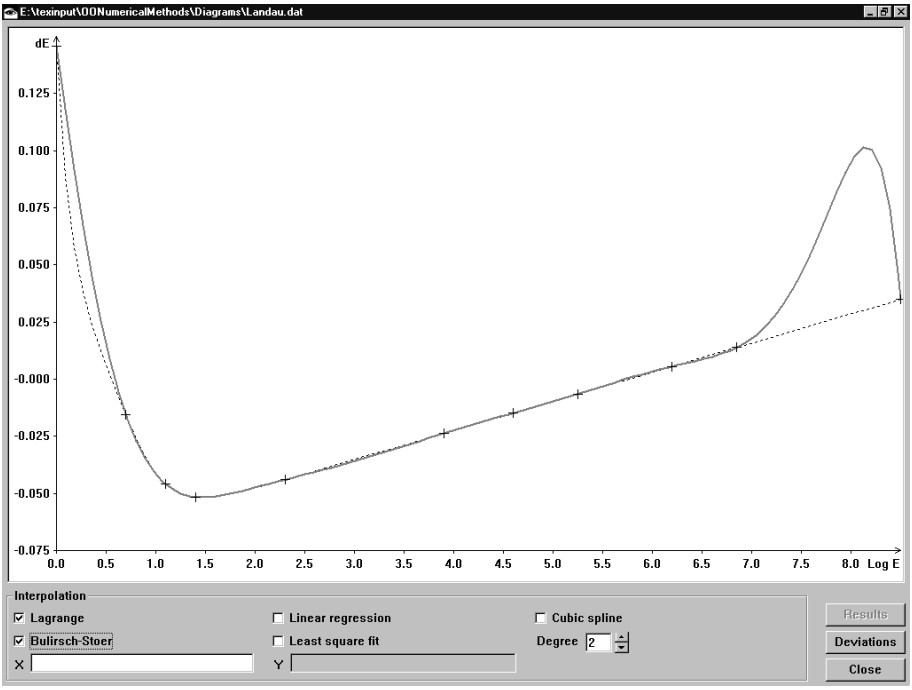


Figure 3-3 Comparison between Lagrange interpolation and interpolation with a rational function

In this case, however, the simulation was based on using experimental data. Contrary to the points of figure 3.1 the points of figure 3.1 are subjected to large fluctuations, because the sample points have been derived from measured data. Thus, interpolation does not work.

As in figure 3.1, the gray thick line is the Lagrange interpolation polynomial and the black dotted line is the cubic spline. Clearly the Lagrange interpolation polynomial is not giving any reasonable interpolation. Cubic spline is not really better as is tries very hard to reproduce the fluctuations of the computed points. In this case, a polynomial fit (*c.f.* section 10.8) is the best choice: the thin black line shows the result of a fit with a 3^{rd} degree polynomial. Another example of unstable interpolation is given in section 10.8 (figure 10-18).

Three implementations of Lagrange interpolation

Once you have verified that a Lagrange interpolation polynomial can be used to perform reliable interpolation over the sample points, you must chose among 3 algorithms to compute the Lagrange interpolation polynomial: direct Lagrange formula, Newton's algorithm and Neville's algorithm.

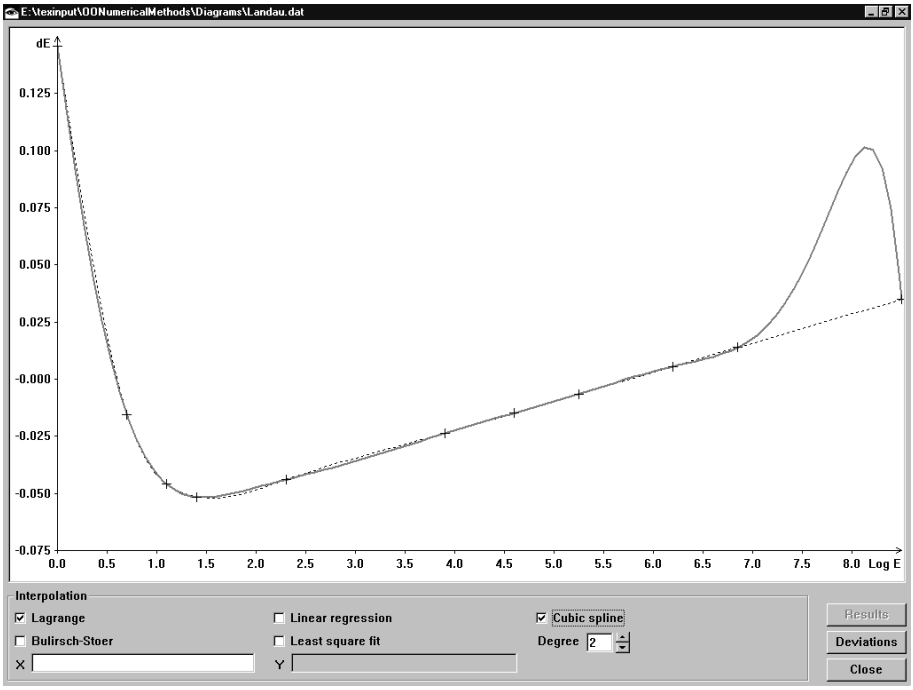


Figure 3-4 Comparison of Lagrange interpolation and cubic spline

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

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

3.2 Lagrange interpolation

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

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

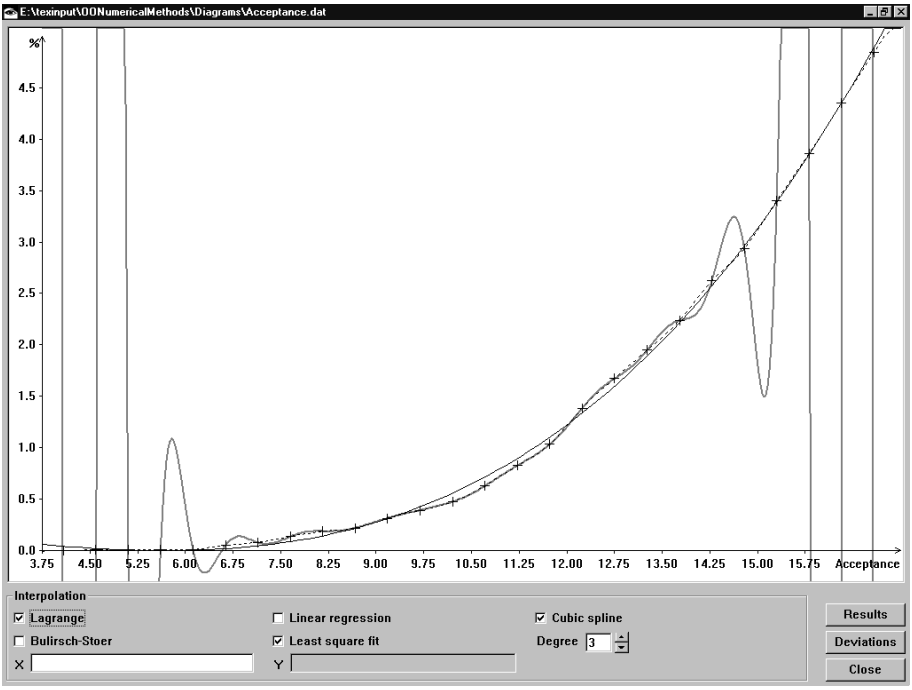


Figure 3-5 Example of misbehaving interpolation

For example, the Lagrange interpolation polynomial of degree 2 on 3 points is given by:

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

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

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

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

Lagrange interpolation — Smalltalk implementation

The object responsible to implement Lagrange interpolation is defined uniquely by the sample points over which the interpolation is performed. In addition

it should behave as a function. In other words it should implement the behavior of a one-variable function as discussed in section 2.2. For example linear interpolation behaves as follows:

Listing 3-6 Linear interpolation

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

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

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

One can also spare the creation of an explicit collection object by implementing these collection methods directly in the Lagrange interpolation class. Now, one can also perform interpolation in the following way:

Listing 3-7 Alternate way to do a linear interpolation

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

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

Listing 3-8 shows the full code of the class implementing the interface shown above.

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

method calls in turn an initialization method, respectively `initialize:` and `initialize`.

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

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

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

Listing 3-8 Smalltalk implementation of the Lagrange interpolation

```
Object subclass: #PMLagrangeInterpolator
   instanceVariableNames: 'pointCollection'
   classVariableNames: ''
   package: 'Math-DHB-Numerical-Math-Interpolator'
```

```
PMLagrangeInterpolator class >> new
    ^super new initialize
```

```
PMLagrangeInterpolator class >> points: aCollectionOfPoints
    ^self new initialize: aCollectionOfPoints
```

```
PMLagrangeInterpolator add: aPoint
    ^pointCollection add: aPoint
```

```
PMLagrangeInterpolator >> defaultSamplePoints
    ^OrderedCollection new
```

```
PMLagrangeInterpolator >> initialize
    ^self initialize: self defaultSamplePoints
```

```
PMLagrangeInterpolator >> initialize: aCollectionOfPoints
    pointCollection := aCollectionOfPoints.
    ^self
```

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

```
PMLagrangeInterpolator >> xPointAt: anInteger
    ^( pointCollection at: anInteger) x
```

```
PMLagrangeInterpolator >> yPointAt: anInteger
    ^( pointCollection at: anInteger) y
```

3.3 Newton interpolation

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

If one writes the Lagrange interpolation polynomial using a generalized Horner expansion, one obtains the Newton's interpolation formula given by [**?**]:

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

The coefficients α_i are obtained by evaluating divided differences as follows:

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

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

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

Newton interpolation — Smalltalk implementation

The object implementing Newton's interpolation algorithm is best implemented as a subclass of the class `PMLagrangeInterpolator` because all methods used to handle the sample points can be reused. This also allows us to keep the interface identical. It has an additional instance variable needed to store the coefficients α_i. Only 4 new methods are needed.

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

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

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

Another reason to keep the method `resetCoefficients` separate is that it must also be called before doing an interpolation if the sample points have been modified directly by the client application after the last interpolation has been made. An alternative is to implement the Observable/Observer pattern so that resetting of the coefficients happens implicitly using events.

However, since modifying the sample points between interpolation should only be a rare occasion when using Newton's algorithm[3] our proposed implementation is much simpler.

Listing 3-9 shows the complete implementation in Smalltalk. The class `NewtonInterpolator` is a subclass of class `LagrangeInterpolator`. The code examples 3-6 and 3-7 can directly be applied to Newton interpolation after replacing the class name `PMLagrangeInterpolator` with `PMNewtonInterpolator`.

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

Listing 3-9 Smalltalk implementation of the Newton interpolation

```
PMLagrangeInterpolator subclass: #PMNewtonInterpolator
   instanceVariableNames: 'coefficients'
   classVariableNames: ''
   package: 'Math-DHB-Numerical-Math-Interpolator'
```

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

```
PMNewtonInterpolator >> computeCoefficients
    | size k1 kn|
    size := pointCollection size.
    coefficients := ( 1 to: size) collect: [ :n | self yPointAt: n].
    1 to: (size - 1)
        do: [ :n |
              size to: ( n + 1)  by: -1
                do: [ :k |
                      k1 := k - 1.
                      kn := k - n.
                      coefficients at: k put: ( (( coefficients at:
                                        k) - ( coefficients at: k1))
                                           / ((self xPointAt: k) -
                                               (self xPointAt:
    kn))).
                    ].
            ].
```

```
PMNewtonInterpolator >> resetCoefficients
    coefficients := nil.
```

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

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

3.4 Neville interpolation

Neville's algorithm uses a successive approximation approach implemented in practice by calculating divided differences recursively. The idea behind the algorithm is to compute the value of the interpolation's polynomials of all degrees between 0 and n. This algorithm assumes that the sample points have been sorted in increasing order of abscissa.

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

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

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

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

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

From the definition above and equation 3.6 one derives a pair of recurrence formulae for the differences:

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

In practice two arrays of differences — one for left and one for right — are allocated. Computation of each order is made within the same arrays. The

differences of the last order can be interpreted as an estimation of the error made in replacing the function by the interpolation's polynomial.

Neville's algorithm is faster than the evaluation of direct Lagrange's interpolation for a small number of points (smaller than about 7[4]. Therefore a simple linear interpolation is best performed using Neville's algorithm. For a large number of points, it becomes significantly slower.

Neville interpolation — Smalltalk implementation

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

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

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

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

Listing 3-10 shows the implementation of Neville's algorithm in Smalltalk. The class `PMNevilleInterpolator` is a subclass of class `PMLagrangeInterpolator`. The code examples 3-6 and 3-7 can directly be applied to Neville interpolation after replacing the class name `PMLagrangeInterpolator` with `PMNevilleInterpolator`. An example of interpolation using the returned estimated error is given in section 6.4.

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

[4] *c.f.* footnote 8 on page 70

Listing 3-10 Smalltalk implementation of Neville's algorithm

```
PMLagrangeInterpolator subclass: #PMNevilleInterpolator
   instanceVariableNames: 'leftErrors rightErrors'
   classVariableNames: ''
   package: 'Math-DHB-Numerical-Math-Interpolator'
```

```
PMNevilleInterpolator >> computeDifference: aNumber at: anInteger1
    order: anInteger2
    | leftDist rightDist ratio |
    leftDist := ( self xPointAt: anInteger1) - aNumber.
    rightDist := (  self xPointAt: ( anInteger1 + anInteger2)) -

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

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

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

```
PMNevilleInterpolator >> value: aNumber
    ^(self valueAndError: aNumber) first
```

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

    ]
                ifFalse:[ error := rightErrors at: ( nearestIndex).
                             nearestIndex := nearestIndex - 1.
                          ].
          answer := answer + error.
        ].
    ^Array with: answer with: error abs
```

3.5 Bulirsch-Stoer interpolation

If the function to interpolate is known to have poles[5] in the vicinity of the real axis over the range of the sample points a polynomial cannot do a good interpolation job [**?**].

In this case it is better to use rational function, that is a quotient of two polynomials as defined hereafter:

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

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

Bulirsch and Stoer have proposed an algorithm for a rational function where $p = \lfloor \frac{n-1}{2} \rfloor$. This means that q is either equal to p if the number of sample points is odd or equal to $p + 1$ if the number of sample points is even. Such

[5] That is, a singularity in the complex plane.

a rational function is called a diagonal rational function. This restriction, of course, limits the type of function shapes that can be interpolated.

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

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

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

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

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

$$\begin{cases} \Delta_{j,i}^{\mathbf{l}_{eft}}(x) &= R_j^i(x) - R_j^{i-1}(x) \\ \Delta_{j,i}^{\mathbf{r}_{ight}}(x) &= R_j^j(x) - R_{j+1}^{i-1}(x) \end{cases} \tag{3.11}$$

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

$$\begin{cases} \Delta_{j,i+1}^{\mathbf{l}_{eft}}(x) &= \dfrac{\frac{x-x_j}{x-x_{i+j+1}}\Delta_{j,i}^{\mathbf{r}_{ight}}(x)\left[\Delta_{j+1,i}^{\mathbf{l}_{eft}}(x) - \Delta_{j,i}^{\mathbf{r}_{ight}}(x)\right]}{\frac{x-x_j}{x-x_{i+j+1}}\Delta_{j,i}^{\mathbf{r}_{ight}}(x) - \Delta_{j+1,i}^{\mathbf{l}_{eft}}(x)} \\ \Delta_{j,i}^{\mathbf{r}_{ight}}(x) &= \dfrac{\Delta_{j+1,i}^{\mathbf{l}_{eft}}(x)\left[\Delta_{j+1,i}^{\mathbf{l}_{eft}}(x) - \Delta_{j,i}^{\mathbf{r}_{ight}}(x)\right]}{\frac{x-x_j}{x-x_{i+j+1}}\Delta_{j,i}^{\mathbf{r}_{ight}}(x) - \Delta_{j+1,i}^{\mathbf{l}_{eft}}(x)} \end{cases} \tag{3.12}$$

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

Bulirsch-Stoer interpolation — Smalltalk implementation

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

Only a single method — the one responsible for the evaluation of the finite differences at each order — must be implemented. All other methods of Neville's interpolation can be reused.

This shows the great power of object-oriented approach. Code written in procedural language cannot be reused that easily. In [**?**] the two codes implementing Neville's and Bulirsch-Stoer interpolation are of comparable length; not surprisingly they also have much in common.

Listing 3-11 shows the implementation of Bulirsch-Stoer interpolation in Smalltalk. The class `PMBulirschStoerInterpolator` is a subclass of class `PMNevilleInterpolator`. The code examples 3-6 and 3-7 can directly be applied to Bulirsch-Stoer interpolation after replacing the class name `PMLagrangeInterpolator` with `PMBulirschStoerInterpolator`.

Listing 3-11 Smalltalk implementation of Bulirsch-Stoer interpolation

```
PMNevilleInterpolator subclass: #PMBulirschStoerInterpolator
   instanceVariableNames: ''
   classVariableNames: ''
   package: 'Math-DHB-Numerical-Math-Interpolator'
```

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

    anInteger1))
                          / ( ratio - ( leftErrors at: ( anInteger1
                                                              +
    1))).
    rightErrors at: anInteger1 put: ( leftErrors at: ( anInteger1 +
                                                        1)) * diff.
    leftErrors at: anInteger1 put: ratio * diff.
```

3.6 Cubic spline interpolation

The Lagrange interpolation polynomial is defined globally over the set of given points and respective function's values. As we have seen in figure 3.1 and to a lesser degree in figure 3.1 Lagrange's interpolation polynomial can have large fluctuations between two adjacent points because the degree of the interpolating polynomial is not constrained.

One practical method for interpolating a set of function's value with a polynomial of constrained degree is to use cubic splines. A cubic spline is a $3^{\mathrm{r}d}$ order polynomial constrained in its derivatives at the end points. A unique cubic spline is defined for each interval between two adjacent points. The interpolated function is required to be continuous up to the second derivative at each of the points.

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

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

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

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

where

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

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

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

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

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

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

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

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

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

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

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

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

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

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

One solves the system of equations 3.18, and possibly 3.19, using direct Gaussian elimination and back substitution (*c.f.* section 8.2). Because the corresponding matrix is tridiagonal, each pivoting step only involves one operation. Thus, resorting to a general algorithm for solving a system of linear equations is not necessary.

Cubic spline interpolation — Smalltalk implementation

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

However, from the behavioral point of view, they are quite similar. Like for Newton interpolation, cubic spline interpolation first needs to compute a series of coefficients, namely the second derivatives, which only depends on the sample points. This calculation only needs to be performed once. Then

the evaluation of the function can be done using equations 3.13, 3.14 and 3.16. Finally, as for the Newton interpolator, any modification of the points requires a new computation of the coefficients. The behavior can be reused from the class `NewtonInterpolator`.

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

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

Natural or constrained cubic spline is flagged independently at each point by testing if the corresponding end-point derivative has been supplied or not. The second derivatives are computed used lazy initialization by the method `computeSecondDerivatives`.

Listing 3-12 shows the implementation of cubic spline interpolation in Smalltalk. The class `PMSplineInterpolator` is a subclass of class `PMNewtonInterpolator`. The code examples 3-6 and 3-7 can directly be applied to cubic spline interpolation after replacing the class name `PMLagrangeInterpolator` with `PMSplineInterpolator`.

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

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

Listing 3-12 Smalltalk implementation of cubic spline interpolation

```
PMNewtonInterpolator subclass: #PMSplineInterpolator
   instanceVariableNames: 'startPointDerivative endPointDerivative'
   classVariableNames: ''
   package: 'Math-DHB-Numerical-Math-Interpolator'
```

```
PMSplineInterpolator >> computeSecondDerivatives
    | size u w s dx inv2dx |
    size := pointCollection size.
    coefficients := Array new: size.
    u := Array new: size - 1.
    startPointDerivative isNil
        ifTrue:
            [coefficients at: 1 put: 0.
            u at: 1 put: 0]
        ifFalse:
            [coefficients at: 1 put: -1 / 2.
            s := 1 / (( self xPointAt: 2) x - ( self xPointAt: 1) x).
            u at: 1
```

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

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

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

```
PMSplineInterpolator >> resetEndPointDerivatives
    self setEndPointDerivatives: ( Array new: 2)
```

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

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

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

3.7 Which method to choose?

At this point some reader might experience some difficulty in choosing among the many interpolation algorithms discussed in this book. There are indeed many ways to skin a cat. Selecting a method depends on what the user intends to do with the data.

First of all, the reader should be reminded that Lagrange interpolation, Newton interpolation and Neville's algorithm are different algorithms computing the values of the same function, namely the Lagrange interpolation polynomial. In other words, the interpolated value resulting from each 3 algorithms is the same (up to rounding errors of course).

The Lagrange interpolation polynomial can be subject to strong variations (if not wild in some cases, figure 3.1 for example) if the sampling points are not

smooth enough. A cubic spline may depart from the desired function if the derivatives on the end points are not constrained to proper values. A rational function can do a good job in cases where polynomials have problems. To conclude, let me give you some rules of thumb to select the best interpolation method based on my personal experience.

If the function to interpolate is not smooth enough, which maybe the case when not enough sampling points are available, a cubic spline is preferable to the Lagrange interpolation polynomial. Cubic splines are traditionally used in curve drawing programs. Once the second derivatives have been computed, evaluation time is of the order of $\mathcal{O}(n)$. You must keep in your mind the limitation[6] imposed on the curvature when using a 3rd order polynomial.

If the Lagrange interpolation polynomial is used to quickly evaluate a tabulated[7] function, Newton interpolation is the algorithm of choice. Like for cubic spline interpolation, the evaluation time is of the order of $\mathcal{O}(n)$ once the coefficients have been computed.

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

Lagrange interpolation can be used for occasional interpolation or when the values over which interpolation is made are changing at each interpolation. The evaluation time of the algorithm is of the order of $\mathcal{O}(n^2)$. Lagrange interpolation is slightly slower than Neville's algorithm as soon as the number of points is larger than 3[8]. However, Neville's algorithm needs to allocate more memory. Depending on the operating system and the amount of available memory the exact place where Lagrange interpolation becomes slower than Neville's algorithm is likely to change.

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

Table 3-13 shows a summary of the discussion. If you are in doubt, I recommend that you make a test first for accuracy and then for speed of execution. Drawing a graph such as in the figures presented in this chapter is quite helpful to get a proper feeling about the possibility offered by various interpolation algorithms on a given set of sample points. If neither Lagrange interpolation nor Bulirsch-Stoer nor cubic spline is doing a good job at interpolating the sample points, you should consider using curve fitting (*c.f.* chapter 10) with an ad-hoc function.

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

[7] A tabulated function is a function, which has been computed at a finite number of its argument.

[8] Such a number is strongly dependent on the operating system and virtual machine. Thus, the reader should check this number him/herself.

Table 3-13 Recommended polynomial interpolation algorithms

*F*eature	*R*ecommended algorithm
Error estimate desired	Neville
Couple of sample points	Lagrange
Medium to large number of sample points	Neville
Many evaluations on fixed sample	Newton
Keep curvature under constraint	Cubic spline
Function hard to reproduce	Bulirsch-Stoer

CHAPTER 4

Iterative algorithms

*C*ent fois sur le métier remettez votre ouvrage.
Nicolas Boileau

When a mathematical function cannot be approximated with a clever expression, such as Lanczos formula introduced in the chapter 2.5, one must resort to compute that function using the integral, the recurrence formula or the series expansion. All these algorithms have one central feature in common: the repetition of the same computation until some convergence criteria is met. Such repetitive computation is called *i*teration.

Figure 4-1 shows the class diagram of the classes discussed in this chapter. This chapter first discusses the implementation of a general-purpose iterative process. Then, we describe a generalization for the finding of a numerical result. Other chapters discuss examples of sub-classing of these classes to implement specific algorithms.

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

4.1 Successive approximations

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

- a set-up phase
- an iteration phase until the result is acceptable

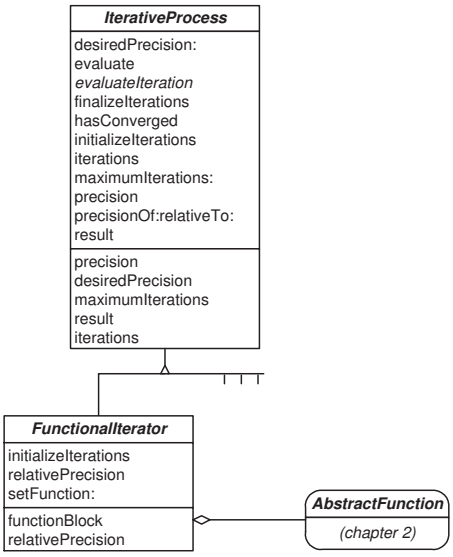


Figure 4-1 Class diagram for iterative process classes

- a clean-up phase

These steps are translated schematically into the flow diagram shown in Figure 4-2.

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

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

During the clean-up phase resources used by the iterative process must be release. In some cases additional results may be derived before leaving the algorithm.

Let us now explicit each of the three stages of the algorithm.

The step computing or choosing an initial object is strongly dependent on the nature of the problem to be solved. In some methods, a good estimate of the solution can be computed from the data. In others using randomly generated objects yields good results. Finally, one can also ask the application's user

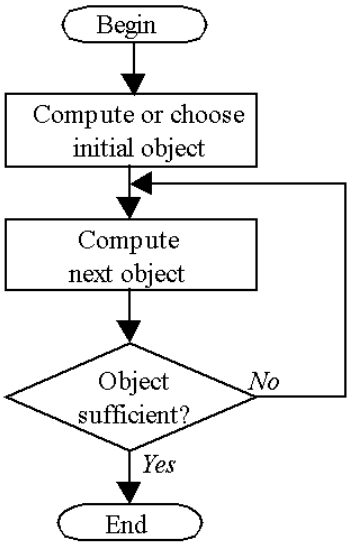


Figure 4-2 Successive approximation algorithm

for directions. In many cases this step is also used to initialize parameters needed by the algorithm.

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

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

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

Design

Now we can add some details to the algorithm. The new details are shown in figure 4-3. This schema allows us to determine the structure of a general object implementing the iterative process. It will be implemented as an abstract class. An abstract class is a class with does not have object instances. A object implementing a specific algorithm is an instance of a particular subclass of the abstract class.

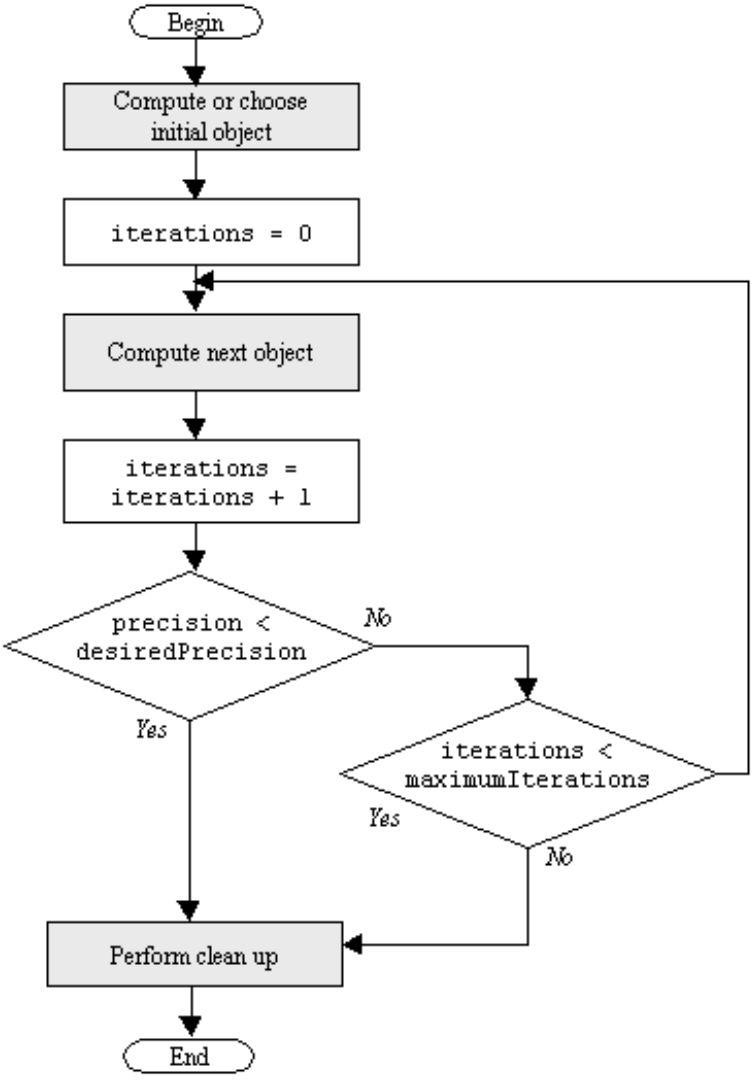


Figure 4-3 Detailed algorithm for successive approximations

The gray boxes in figure 4-3 represent the methods, which must be implemented explicitly by the subclass. The abstract class calls them. However, the exact implementation of these methods is not defined at this stage. Such methods are called *h*ook methods.

Using this architecture the abstract class is able to implement the iterative process without any deep knowledge of the algorithm. Algorithm specific methods are implemented by the subclass of the abstract class.

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

- `iterations`keeps track of the number of iterations, that is the number of successive approximations,
- `maximumIterations` maximum number of allowed iterations,
- `desiredPrecision` the precision to attain, that is, how close to the solution should the solution object be when the algorithm is terminated,
- `precision` the precision achieved by the process. Its value is updated after each iteration and it is used to decide when to stop.

The methods of the class `IterativeProcess` are shown in figure 4-4 in correspondence with the general execution flow shown in figure 4-3. The two methods `initializeIterations` and `finalizeIterations` should be implemented by the subclass but the abstract class provides a default behavior: doing nothing. The method `evaluateIteration` must be implemented by each subclass.

Since the precision of the last iteration is kept in an instance variable, the method `hasConverged` can be called at any time after evaluation, thus providing a way for client classes to check whether the evaluation has converged or not.

Iterative process — Smalltalk implementation

Even though we are dealing for the moment with an abstract class we are able to present a scenario of use illustrating the public interface to the class. Here is how a basic utilization of an iterative process object would look like.

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

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

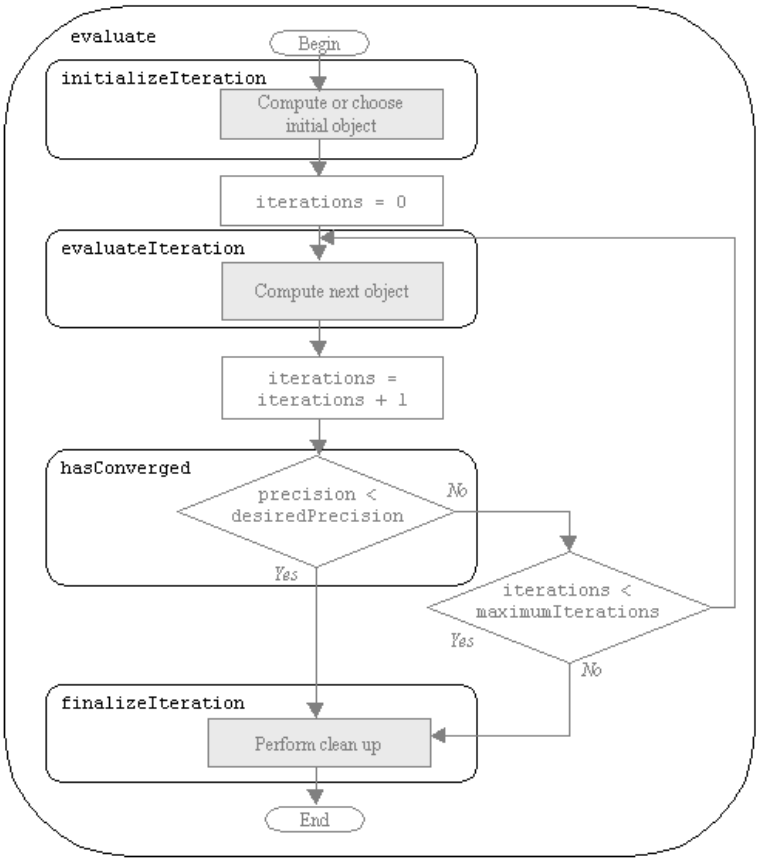


Figure 4-4 Methods for successive approximations

To give the user a possibility to have more control, one can extend the public interface of the object to allow defining the parameters of the iterative process: the desired precision and the maximum number of iterations. In addition, the user may want to know the precision of the attained result and the number of iterations needed to obtain the result. The following code sample shows an example of use for all public methods defined for an iterative process. The precision of the attained result and the number of iterations are printed on the transcript window.

```
| iterativeProcess result precision |
iterativeProcess := <a subclass of DhbIterativeProcess> new.
iterativeProcess desiredPrecision: 1.0e-6; maximumIterations: 25.
result := iterativeProcess evaluate.
iterativeProcess hasConverged
   ifTrue: [ Transcript nextPutAll: 'Result obtained after '.
```

```
            iterativeProcess iteration printOn: Transcript.
            Transcript nextPutAll: 'iterations. Attained precision
    is'.
            iterativeProcess precision printOn: Transcript.
            ]
    ifFalse:[ Transcript nextPutAll: 'Process did not converge'.].
Transcript cr.
```

Listing 4-5 shows the Smalltalk implementation of the iterative process.

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

Default values for the desired precision and the maximum number of iterations are kept in class methods for easy editing. The method initialize loads these default values for each newly created instance. The default precision is set to the machine precision discussed in section 1.3.

The methods used to modify the desired precision (`desiredPrecision:`) and the maximum number of iterations (`maximumIterations:`) check the value to prevent illegal definitions, which could prevent the algorithm from terminating.

Since there is no explicit declaration of abstract class and abstract methods in Smalltalk[1] the three methods `initializeIterations`, `evaluateIteration` and `finalizeIterations`, are implemented with a reasonable default behavior. The methods `initializeIterations` and `finalizeIterations` do nothing. The method `evaluateIteration` calls the method `subclassResponsibility`, which raises an exception when called. Using this technique is the Smalltalk way of creating an abstract method.

Listing 4-5 Smalltalk implementation of an iterative process

```
Object subclass: #PMIterativeProcess
   instanceVariableNames: 'precision desiredPrecision
    maximumIterations result iterations'
   classVariableNames: ''
   package: 'Math-Core-Process'
```

```
PMIterativeProcess class >> defaultMaximumIterations
    ^50
```

```
PMIterativeProcess class >> defaultPrecision
    ^PMFloatingPointMachine new defaultNumericalPrecision
```

[1] An abstract class is a class containing at least an abstract method; an abstract method contains the single conventional statement: `self subclassResponsibility`

```
PMIterativeProcess >> desiredPrecision: aNumber
    aNumber > 0
        ifFalse: [ ^self error: 'Illegal precision: ', aNumber

    printString].
    desiredPrecision := aNumber.
```

```
PMIterativeProcess >> evaluate
    iterations := 0.
    self initializeIterations.
    [iterations := iterations + 1.
    precision := self evaluateIteration.
    self hasConverged or: [iterations >= maximumIterations]]
            whileFalse: [].
    self finalizeIterations.
    ^self result
```

```
PMIterativeProcess >> evaluateIteration
    ^self subclassResponsibility
```

```
PMIterativeProcess >> finalizeIterations
   "Perform cleanup operation if needed (must be implemented by
    subclass)."
```

```
PMIterativeProcess >> hasConverged
    ^precision <= desiredPrecision
```

```
PMIterativeProcess >> initialize
    desiredPrecision := self class defaultPrecision.
    maximumIterations := self class defaultMaximumIterations.
    ^self
```

```
PMIterativeProcess >> initializeIterations
   "Initialize the iterations (must be implemented by subclass when
    needed)."
```

```
PMIterativeProcess >> iterations
    ^iterations
```

```
PMIterativeProcess >> maximumIterations: anInteger
    ( anInteger isInteger and: [ anInteger > 1])
        ifFalse: [ ^self error: 'Invalid maximum number of iteration:
                                            ', anInteger
    printString].
    maximumIterations := anInteger
```

```
PMIterativeProcess >> precision
    ^precision
```

```
PMIterativeProcess >> precisionOf: aNumber1 relativeTo: aNumber2
    ^aNumber2 > PMFloatingPointMachine new defaultNumericalPrecision
        ifTrue: [ aNumber1 / aNumber2]
        ifFalse:[ aNumber1]
```

```
PMIterativeProcess >> result
   ^result
```

> Note: The method `precisionOf:relativeTo:` implements the computation of the relative precision. This is discussed in section 4.2.

4.2 Evaluation with relative precision

So far we have made no assumption about the nature of the solution searched by an iterative process. In this section we want to discuss the case when the solution is a numerical value.

As discussed in section 1.3 a floating-point number is a representation with constant relative precision. It is thus meaningless to use absolute precision to determine the convergence of an algorithm. The precision of an algorithm resulting in a numerical value ought to be determined relatively.

One way to do it is to have the method `evaluateIteration` returning a relative precision instead of an absolute number. Relative precision, however, can only be evaluated if the final result is different from zero. If the result is zero, the only possibility is to check for absolute precision. Of course, in practice one does not check for equality with zero. The computation of a relative precision is carried only if the absolute value of the result is larger than the desired precision.

The reasoning behind the computation of the relative error is quite general. Thus, a general-purpose class `FunctionalIterator` has been created to implement a method computing the relative precision from an absolute precision and a numerical result. In addition, since all subclasses of `FunctionalIterator` use a function a general method to handle the definition of that function is also supplied.

Relative precision — Smalltalk implementation

In this case the public interface is extended with a creation method taking as argument the function on which the process operates. The code example of section 4.1 then becomes:

```
| iterativeProcess result |
iterativeProcess := <a subclass of PMFunctionalIterator>
                  function: ( PMPolynomial coefficients: #(1 2 3)).
result := iterativeProcess evaluate.
iterativeProcess hasConverged ifFalse:[ <special case processing>].
```

In this example the function on which the process will operate is the polynomial $3x^2 + 2x + 1$ (*c.f.* section 2.3).

Listing **??** shows the implementation of the abstract class `PMFunctionalIterator` in Smalltalk.

This class has one instance variable `functionBlock` to store the function. A single class method allows creating a new instance while defining the function.

As we have seen in section 2.2, a function can be any object responding to the message value:. This allows supplying any block of Smalltalk code as argument to the constructor method. However, the user can also supply a class implementing the computation of the function with a method with selector `value:`. For example, an instance of the class `PMPolynomial` discussed in section **??** can be used.

The instance method `setFunction:` is used to set the instance variable `functionBlock`. In order to prevent a client class from sending the wrong object, the method first checks whether the supplied object responds to the message `value:`. This is one way of ensuring that the arguments passed to a method conform to the expected protocol. This way of doing is only shown as an example, however. It is not recommend in practice. The responsibility of supplying the correct arguments to a Smalltalk method is usually the responsibility of the client class.

The method `initializeIterations` first checks whether a function block has been defined. Then it calls the method `computeInitialValues`. This method is a hook method, which a subclass must implement to compute the value of the result at the beginning of the iterative process.

The computation of relative precision is implemented at two levels. One general method, `precisionOf:relativeTo:`, implemented by the superclass allows the computation of the relative precision relative to any value. Any iterative process can use this method. The method `relativePrecision` implements the computation of the precision relative to the current result.

Listing 4-6 Smalltalk implementation of the class `PMFunctionalIteration`

```
PMIterativeProcess subclass: #PMFunctionalIterator
   instanceVariableNames: 'functionBlock'
   classVariableNames: ''
   package: 'Math-DHB-Numerical-Math-FunctionIterator'
```

```
PMFunctionalIterator class >> function: aBlock
    ^self new setFunction: aBlock; yourself
```

```
PMFunctionalIterator >> initializeIterations
    functionBlock isNil ifTrue: [self error: 'No function supplied'].
    self computeInitialValues
```

```
PMFunctionalIterator >> relativePrecision: aNumber
    ^self precisionOf: aNumber relativeTo: result abs
```

```
PMFunctionalIterator >> setFunction: aBlock
    ( aBlock respondsTo: #value:)
        ifFalse:[ self error: 'Function block must implement the
                                                        method
    value:'].
    functionBlock := aBlock
```

4.3 Examples

As we have dealt with abstract classes, this chapter did not give concrete examples of use. By consulting the rest of this book the reader will find numerous examples of subclasses of the two classes described in this chapter. Table 4-7 lists the sections where each algorithm using the iterative process framework is discussed.

Table 4-7 Algorithms using iterative processes

Algorithm or class of algorithm	Superclass	Chapter or section
Zero finding	Function iterator	Chapter 5
Integration	Function iterator	Chapter 6
Infinite series and continued fractions	Function iterator	Chapter 7
Matrix eigenvalues	Iterative process	Section 8.6
Non-linear least square fit	Iterative process	Section 10.9
Maximum likelihood fit	Iterative process	Section 10.10
Function minimization	Function iterator	Chapter 11
Cluster analysis	Iterative process	Section ??

CHAPTER 5

Finding the zero of a function

Le zéro, collier du néant.[1]
Jean Cocteau

The zeroes of a function are the values of the function's variable for which the value of the function is zero. Mathematically, given the function $f(x)$, z is a zero of when $f(z) = 0$. This kind of problem is can be extended to the general problem of computing the value of the inverse function, that is finding a value x such that $f(x) = c$ where c is a given number. The inverse function is noted as $f^{-1}(x)$. Thus, one wants to find the value of $f^{-1}(c)$ for any c. The problem can be transformed into the problem of finding the zero of the function $\tilde{f}(x) = f(x) - c$.

The problem of finding the values at which a function takes a maximum or minimum value is called searching for the extremes of a function. This problem can be transformed into a zero-finding problem if the derivative of the function can be easily computed. The extremes are the zeroes of the function's derivative.

Figure 5-1 shows the class diagram of the classes discussed in this chapter.

5.1 Introduction

Let us begin with a concrete example.

Often an experimental result is obtained by measuring the same quantity several times. In scientific publications, such a result is published with two numbers: the average and the standard deviation of the measurements. This is

[1] The zero, a necklace for emptiness.

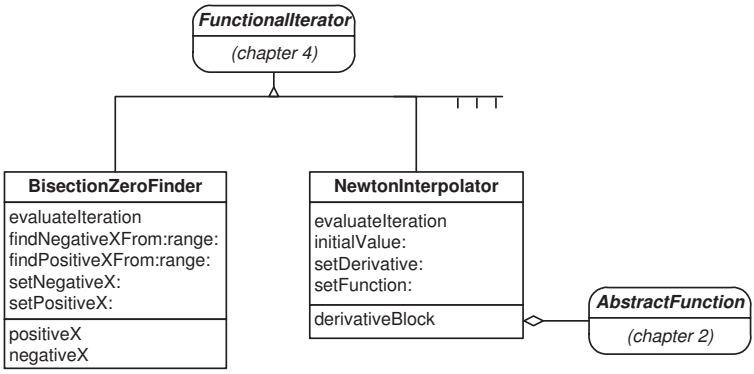


Figure 5-1 Class diagram for zero finding classes

true for medical publication as well. As we have already discussed in section 2.4, obstetricians prefer to think in terms of risk and prefer to use centiles instead of average and standard deviation. Assuming that the measurements were distributed according to a normal distribution (*c.f.* section 9.6), the 90th centile is the solution to the following equation:

$$\text{erf}\,(x) = 0.9 \tag{5.1}$$

That is, we need to find the zero of the function $f\,(x) = \text{erf}\,(x) - 0.9$. The answer is $x = 1.28$ with a precision of two decimals. Thus, if μ and σ are respectively the average and standard deviation of a published measurement, the 90th centile is given by $\mu + 1.28 \cdot \sigma$. Using equation 2.16 the 10th centile is given by $\mu - 1.28 \cdot \sigma$.

5.2 Finding the zeroes of a function — Bisection method

Let assume that one knows two values of x for which the function takes values of opposite sign. Let us call x_{pos} the value such that $f\,(x_{pos}) > 0$ and x_{neg} the value such that $f\,(x_{neg}) < 0$. If the function is continuous between x_{pos} and x_{neg}, there exists at least one zero of the function in the interval $[x_{pos}, x_{neg}]$. This is illustrated in figure 5-2. If the function f is not continuous over the interval where the sign of the function changes, then the presence of a zero cannot be guaranteed[2]. The continuity requirement is essential for the application of the bisection algorithm.

The values x_{pos} and x_{neg} are the initial values of the bisection algorithm. The algorithm goes as follows:

1. Compute $x = \frac{x_{pos} - x_{neg}}{2}$.

[2]The inverse function is such an example. It changes sign over 0 but has no zeroes for any finite x

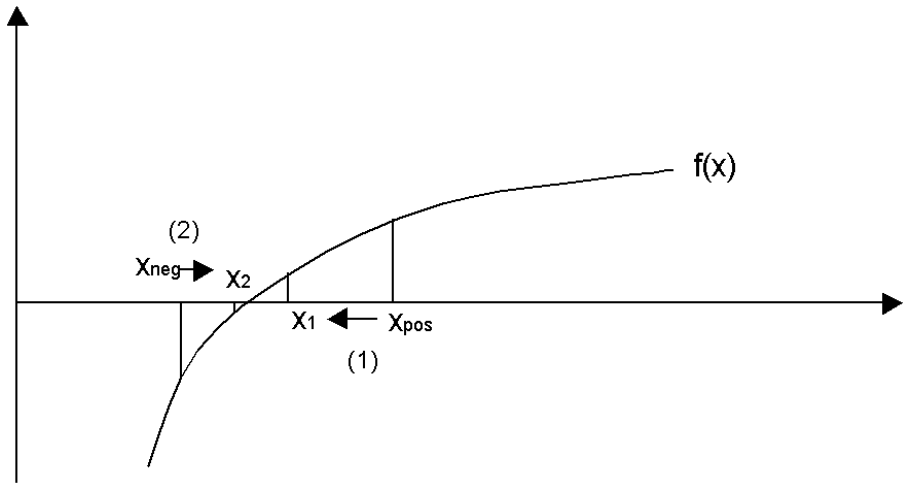


Figure 5-2 The bisection algorithm

2. If $f(x) > 0$, set $x_{pos} = x$ and goto step 4.
3. Otherwise set $x_{neg} = x$.
4. If $|x_{pos}, x_{neg}| > \epsilon$ go back to step 1. ϵ is the desired precision of the solution.

The first couple of steps of the bisection algorithm are represented geometrically on figure 5-2. Given the two initial values, x_{pos} and x_{neg}, the first iteration of the algorithm replaces x_{pos} with x_1. The next step replaces x_{neg} with x_2.

For a given pair of initial values, x_{pos} and x_{neg}, the number of iterations required to attain a precision ϵ is given by:

$$n = \left\lceil \log_2 \frac{|x_{pos}, x_{neg}|}{\epsilon} \right\rceil . \tag{5.2}$$

For example if the distance between the two initial values is 1 the number of iterations required to attain a precision of 10^{-8} is 30. It shows that the bisection algorithm is rather slow.

Knowledge of the initial values, x_{pos} and x_{neg}, is essential for starting the algorithm. Methods to define them must be supplied. Two convenience methods are supplied to sample the function randomly over a given range to find each initial value. The random number generator is discussed in section 9.4.

The bisection algorithm is a concrete implementation of an iterative process. In this case, the method `evaluateIteration` of figure 4-4 implements steps 2, 3 and 4. The precision at each iteration is $|x_{pos} - x_{neg}|$ since the zero of the function is always inside the interval defined by x_{pos} and x_{neg}.

Bisection algorithm — Smalltalk implementation

The class of the object implementing the bisection algorithm is a subclass of the abstract class `PMFunctionalIterator`. The class `PMBisectionZeroFinder` needs the following additional instance variables:

- `positiveX` x_{pos} and
- `negativeX` x_{neg}

The bisection algorithm proper is implemented only within the method `evaluateIteration`. Other necessary methods have already been implemented in the iterative process class.

Finding the zero of a function is performed by creating an instance of the class `PMBisectionZeroFinder` and giving the function as the argument of the creation method as explained in section 4.2. For example the following code finds the solution of equation 5.1.

```
| zeroFinder result |
zeroFinder:= PMBisectionZeroFinder function: [ :x | x errorFunction
    - 0.9].
zeroFinder setPositiveX: 10; setNegativeX: 0.
result := zeroFinder evaluate. zeroFinder
hasConverged
   ifFalse:[ <special case processing>].
```

The second line creates the object responsible to find the zero. The third line defines the initial values, x_{pos} and x_{neg}. The fourth line performs the algorithm and stores the result if the algorithm has converged. The last two lines check for convergence and take corrective action if the algorithm did not converge.

Listing 5-3 shows the implementation of the bisection zero finding algorithm in Smalltalk.

The class `PMBisectionZeroFinder` is a subclass of the class `PMFunctionalIterator`. As one can see only a few methods need to be implemented. Most of them pertain to the definition of the initial interval. In particular, convenience methods are supplied to find a positive and negative function value over a given interval.

The methods defining the initial values, x_{pos} and x_{neg}, are `setPositiveX:` and `setNegativeX:` respectively. An error is generated in each method if the function's value does not have the proper sign. The convenience methods to find random starting values are respectively `findPositiveXFrom:range:` and `findNegativeXFrom:range:`. The method `computeInitialValues` does not compute the initial values. Instead it makes sure that x_{pos} and x_{neg} have been properly defined.

Listing 5-3 Smalltalk implementation of the bisection algorithm

```
PMFunctionalIterator subclass: #PMBisectionZeroFinder
   instanceVariableNames: 'positiveX negativeX'
   classVariableNames: ''
   package: 'Math-DHB-Numerical-Math-FunctionIterator'
```

```
PMBisectionZeroFinder >> computeInitialValues
    positiveX isNil
        ifTrue: [ self error: 'No positive value supplied'].
    negativeX isNil
        ifTrue: [ self error: 'No negative value supplied'].
```

```
PMBisectionZeroFinder >> evaluateIteration
    result := ( positiveX + negativeX) * 0.5.
    ( functionBlock value: result) > 0
        ifTrue: [ positiveX := result]
        ifFalse:[ negativeX := result].
    ^self relativePrecision: ( positiveX - negativeX) abs
```

```
PMBisectionZeroFinder >> findNegativeXFrom: aNumber1 range: aNumber2
    | n |
    n := 0.
    [ negativeX := Number random * aNumber2 + aNumber1.
      ( functionBlock value: negativeX) < 0
        ] whileFalse: [ n := n + 0.1.
                        n > maximumIterations
                            ifTrue: [ self error: 'Unable to find a
                                                  negative function
    value'].
                    ].
```

```
PMBisectionZeroFinder >> findPositiveXFrom: aNumber1 range: aNumber2
    | n |
    n := 0.
    [ positiveX := Number random * aNumber2 + aNumber1.
      ( functionBlock value: positiveX) > 0
        ] whileFalse: [ n := n + 1.
                        n > maximumIterations
                            ifTrue: [ self error: 'Unable to find a
                                                  positive function
    value'].
                    ]
```

```
PMBisectionZeroFinder >> setNegativeX: aNumber
    ( functionBlock value: aNumber) < 0
        ifFalse:[ self error: 'Function is not negative at x = ',
                                                  aNumber
    printString].
    negativeX := aNumber.
```

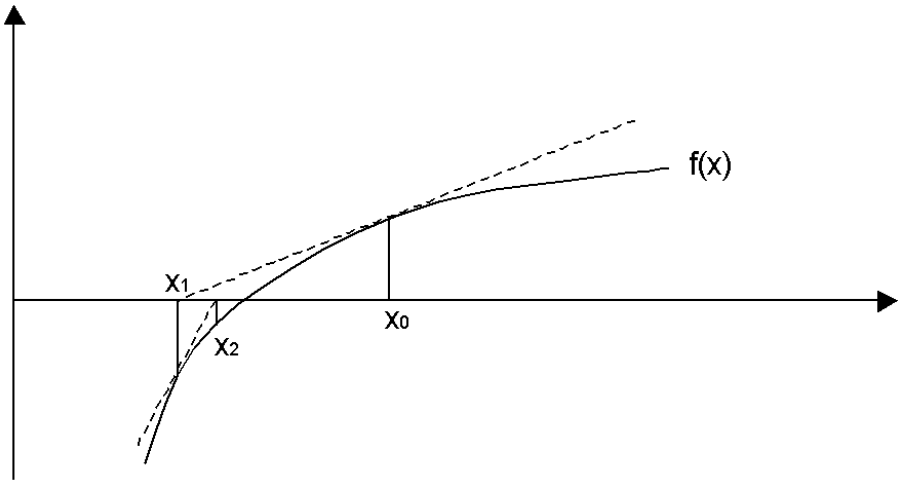


Figure 5-4 Geometrical representation of Newton's zero finding algorithm

```
PMBisectionZeroFinder >> setPositiveX: aNumber
    ( functionBlock value: aNumber) > 0
        ifFalse:[ self error: 'Function is not positive at x = ',
                                                    aNumber
    printString].
    positiveX := aNumber
```

5.3 Finding the zero of a function — Newton's method

Isaac Newton has designed an algorithm working by successive approximations[?]. Given a value x_0 chosen in the vicinity of the desired zero, the following series:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, \tag{5.3}$$

where $f'(x)$ is the first derivative of $f(x)$, converges toward a zero of the function. This algorithm is sometimes called Newton-Ralphson[?].

Figure 5-4 shows the geometrical interpretation of the series. $f'(x)$ is the slope of the tangent to the curve of the function $f(x)$ at the point x_n. The equation of this tangent is thus given by:

$$y = (x - x_n) \cdot f'(x_n) + f(x_n) \tag{5.4}$$

One can then see that x_{n+1} is the point where the tangent to the curve at the point x_n crosses the x-axis. The algorithm can be started at any point where the function's derivative is non- zero.

The technique used in Newton's algorithm is a general technique often used in approximations. The function is replaced by a linear approximation[3], that is a straight line going through the point defined by the preceding value and its function's value. The slope of the straight line is given by the first derivative of the function. The procedure is repeated until the variation between the new value and the preceding one is sufficiently small. We shall see other examples of this technique in the remainder of this book (*c.f.* sections 10.9, 10.10 and 11.1).

From equation 5.3, one can see that the series may not converge if $f'(x)$ becomes zero. If the derivative of the function is zero in the vicinity of the zero, the bisection algorithm gives better results. Otherwise Newton's algorithm is highly efficient. It usually requires 5-10 times less iteration than the bisection algorithm. This largely compensates for the additional time spent in computing the derivative.

The class implementing Newton's algorithm belongs to a subclass of the functional iterator described in section 4.2. An additional instance variable is needed to store the function's derivative.

Newton's method — Smalltalk implementation

Listing 5-5 shows the complete implementation in Smalltalk. The class `PMNewtonZeroFinde` is a subclass of the class `PMFunctionalIterator` described in section 4.2. For example the following code finds the solution of equation 5.1.

```
| zeroFinder result |
zeroFinder:= PMNewtonZeroFinder
            function: [ :x | x errorFunction - 0.9]
            derivative: [ :x | PMErfApproximation new normal: x].
zeroFinder initialValue: 1.
result := zeroFinder evaluate.
zeroFinder hasConverged
   ifFalse:[ <special case processing>].
```

The second line creates the object responsible to find the zero supplying the function and the derivative[4]. The third line defines the starting value. The fourth line performs the algorithm and stores the result if the algorithm has converged. The last two lines check for convergence and take corrective action if the algorithm did not converge.

The method `computeInitialValues` is somewhat complex. First, it checks whether the user supplied an initial value. If not, it is assigned to 0. Then the method checks whether the user supplied a derivative. If not a default deriva-

[3] Mathematically, this corresponds to estimate the function using the first two terms of its Taylor series.

[4] As we have seen in section 2.4, the normal distribution is the derivative of the error function.

tive function is supplied as a block closure by the method `defaultDerivativeBlock`. The supplied block closure implements the formula of equation 5.5.

$$\frac{df\left(x\right)}{dx}=\lim_{\epsilon\to 0}\frac{f\left(x+\epsilon\right)-f\left(x-\epsilon\right)}{2\epsilon}. \tag{5.5}$$

If a derivative is supplied, it is compared to the result of the derivative supplied by default. This may save a lot of trouble if the user made an error in coding the derivative. Not supplying a derivative has some negative effect on the speed and limits the precision of the final result. The method `initializeIterations` also checks whether the derivative is nearly zero for the initial value. If that is the case, the initial value is changed with a random walk algorithm. If no value can be found such that the derivative is non-zero an error is generated.

If the function is changed, the supplied derivative must be suppressed. Thus, the method `setFunction:` must also force a redefinition of the derivative. A method allows defining the initial value. A creation method defining the function and derivative is also supplied for convenience.

Like for the bisection, the algorithm itself is coded within the method `evaluateIteration`. Other methods needed by the algorithm have been already implemented in the superclasses.

Listing 5-5 Smalltalk implementation of Newton's zero-finding method

5.4 Example of zero-finding — Roots of polynomials

The zeroes of a polynomial function are called the roots of the polynomial. A polynomial of degree n has at most n real roots. Some[5] of them maybe complex, but are not covered in this book.

If x_0 is a root of the polynomial $P\left(x\right)$, then $P\left(x\right)$ can be exactly divided by the polynomial $x-x_0$. In other words there exists a polynomial $P_1\left(x\right)$ such that:

$$P\left(x\right)=\left(x-x_0\right)\cdot P_1\left(x\right) \tag{5.6}$$

Equation 5.6 also shows that all roots of $P_1\left(x\right)$ are also roots of $P\left(x\right)$. Thus, one can carry the search of the roots using recurrence. In practice a loop is more efficient[6]. The process is repeated at most n times and will be interrupted if a zero finding step does not converge.

One could use the division algorithm of section 2.3 to find $P_1\left(x\right)$. In this case, however, the inner loop of the division algorithm — that is, the loop over the coefficients of the dividing polynomial — is not needed since the dividing

[5] If the degree of the polynomial is odd, there is always at least one non-complex root. Polynomials of even degree may have only complex roots and no real roots.

[6] The overhead comes from allocating the structures needed by the method in each call.

polynomial has only two terms. In fact, one does not need to express $x - x_0$ at all as a polynomial. To carry the division one uses a specialized algorithm taking the root as the only argument. This specialized division algorithm is called *d*eflation [**?**].

Polynomials are very smooth so Newton's algorithm is quite efficient for finding the first root. To ensure the best accuracy for the deflation it is recommended to find the root of smallest absolute value first. This works without additional effort since our implementation of Newton's algorithm uses 0 at the starting point by default. At each step the convergence of the zero-finder is checked. If a root could not be found the process must be stopped. Otherwise, the root finding loop is terminated when the degree of the deflated polynomial becomes zero.

Roots of polynomials — Smalltalk implementation

Roots of a polynomial can be obtained as an `OrderedCollection`. For example, the following code sample retrieves the roots of the polynomial $x^3 - 2x^2 - 13x - 10$:

```
(PMPolynomial coefficients: #(-10 -13 -2 1)) roots.
```

returns: `an OrderedCollection(-0.9999999999999998 -2.0 5.0)`
The methods needed to get the roots are shown in Listing 5-6.

The deflation algorithm is implemented in the method `deflateAt:` using the iterator method `collect:` (*c.f.* section **??**). An instance variable is keeping track of the remainder of the division within the block closure used by the method `collect:`.

The roots are kept in an `OrderedCollection` object constructed in the method `roots:`. The size of the `OrderedCollection` is initialize to the maximum expected number of real roots. Since some of the roots may be complex, we are storing the roots in an `OrderedCollection`, instead of an `Array`, so that the number of found real roots can easily be obtained. This method takes as argument the desired precision used in the zero finding algorithm. A method root uses the default numerical machine precision as discussed in section 1.4.

Listing 5-6 Smalltalk implementation of finding the roots of a polynomial

```
PMPolynomial >> deflatedAt: aNumber
    | remainder next newCoefficients|
    remainder := 0.
    newCoefficients := coefficients collect:
                        [ :each |
                          next := remainder.
                          remainder := remainder * aNumber + each.
                          next].
    ^self class new: ( newCoefficients copyFrom: 2 to:
                                        newCoefficients size)
     reverse
```

```
PMPolynomial >> roots
    ^self roots: PMFloatingPointMachine new

    defaultNumericalPrecision
```

```
PMPolynomial >> roots: aNumber
    | pol roots x rootFinder |
    rootFinder := PMNewtonZeroFinder new.
    rootFinder desiredPrecision: aNumber.
    pol := self class new: ( coefficients reverse collect: [ :each |
                                                          each
    asFloat]).
    roots := OrderedCollection new: self degree.
    [ rootFinder setFunction: pol; setDerivative: pol derivative.
      x := rootFinder evaluate.
      rootFinder hasConverged
        ] whileTrue: [ roots add: x.
                       pol := pol deflatedAt: x.
                       pol degree > 0
                         ifFalse: [ ^roots].
                     ].
    ^roots
```

5.5 Which method to choose

There are other zero-finding techniques: *r*egula falsi, Brent [**?**]. For each of these methods, however, a specialist of numerical methods can design a function causing that particular method to fail.

In practice the bisection algorithm is quite slow as can be seen from equation 5.2. Newton's algorithm is faster for most functions you will encounter. For example, it takes 5 iterations to find the zero of the logarithm function with Newton's algorithm to a precision of $3 \cdot 10^{-9}$ whereas the bisection algorithm requires 29 to reach a similar precision. On the other hand bisection is rock solid and will always converge over an interval where the function has no singularity. Thus, it can be used as a recovery when Newton's algorithm fails.

My own experience is that Newton's algorithm is quite robust and very fast. It should suffice in most cases. As we have seen Newton's algorithm will fail if it encounters a value for which the derivative of the function is very small. In this case, the algorithm jumps far away from the solution. For these cases, the chances are that the bisection algorithm will find the solution if there is any. Thus, combining Newton's algorithm with bisection is the best strategy if you need to design a foolproof algorithm.

Implementing an object combining both algorithms is left as an exercise to the reader. Here is a quick outline of the strategy to adopt. Newton's algorithm must be modified to keep track of values for which the function takes

negative values and positive values — that is the values x_{pos} and x_{neg} — making sure that the value $|x_{pos} - x_{neg}|$ never increases. Then, at each step, one must check that the computed change does not cause the solution to jump outside of the interval defined by x_{pos} and x_{neg}. If that is the case, Newton's algorithm must be interrupted for one step using the bisection algorithm.

CHAPTER 6

Integration of functions

Les petits ruisseaux font les grandes rivières[1]
French proverb

Many functions are defined by an integral. For example, the three functions discussed in the last 3 sections of chapter 2 were all defined by an integral. When no other method is available the only way to compute such function is to evaluate the integral. Integrals are also useful in probability theory to compute the probability of obtaining a value over a given interval. This aspect will be discussed in chapter 9. Finally integrals come up in the computation of surfaces and of many physical quantities related to energy and power. For example, the power contained in an electromagnetic signal is proportional to the integral of the square of the signal's amplitude.

The French proverb quoted at the beginning of this chapter is here to remind people that an integral is defined formally as the infinite sum of infinitesimal quantities.

6.1 Introduction

Let us begin with a concrete example. This time we shall take a problem from physics 101.

When light is transmitted through a narrow slit, it is diffracted. The intensity of the light transmitted at an angle ϑ, $I(\vartheta)$, is given by:

$$I(\vartheta) = \frac{\sin^2 \vartheta}{\vartheta^2} \tag{6.1}$$

[1] Small streams build great rivers.

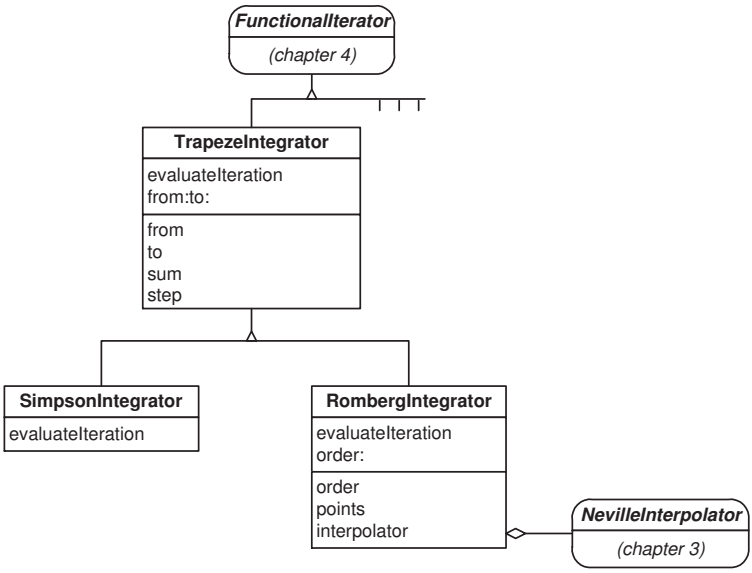


Figure 6-1 Class diagram of integration classes

If one wants to compute the fraction of light which is transmitted within the first diffraction peak, one must compute the expression:

$$I(\vartheta) = \frac{1}{\pi}\int_{-\pi}^{\pi} \frac{\sin^2 \vartheta}{\vartheta^2} d\vartheta. \tag{6.2}$$

The division by π is there because the integral of $I(\vartheta)$ from $-\infty$ to $+\infty$ is equal to π. No closed form exists for the integral of equation 6.2: it must be computed numerically. This answer is 90.3%.

In this chapter we introduce 3 integration algorithms. Figure 6-1 shows the corresponding class diagram. The first one, trapeze integration, is only introduced for the sake of defining a common framework for the next two algorithms: Simpson and Romberg integration. In general, the reader should use Romberg's algorithm. It is fast and very precise. There are, however, some instances where Simpson's algorithm can be faster if high accuracy is not required.

6.2 General framework — Trapeze integration method

Let us state it at the beginning. One should not use the trapeze integration algorithm in practice. The interest of this algorithm is to define a general framework for numerical integration. All subclasses of the class responsible for implementing the trapeze integration algorithm will reuse most the mechanisms described in this section.

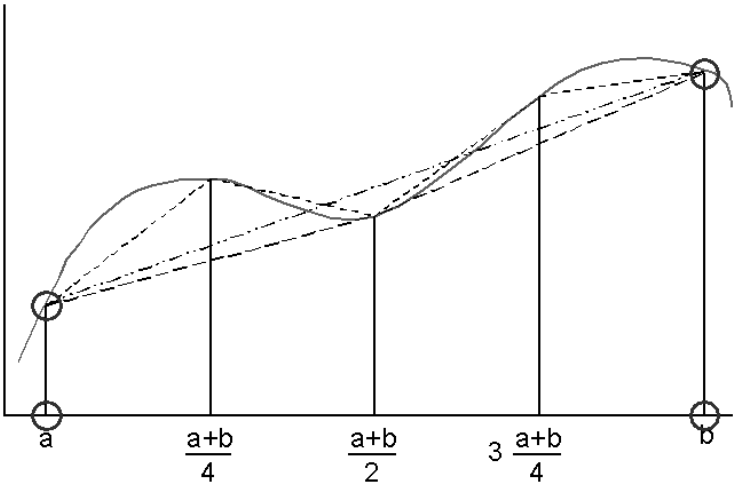


Figure 6-2 Geometrical interpretation of the trapeze integration method

The trapeze numerical integration method takes its origin in the series expansion of an integral. This series expansion is expressed by the Euler-Maclaurin formula shown hereafter [**?**]:

$$\int_a^b f(x)\,dx = \frac{b-a}{2}\left[f(a)+f(b)\right] - \sum_n \frac{(b-a)^2}{(2n)!} B_{2n}\left[\frac{d^{2n-1}f(b)}{dx^{2n-1}} - \frac{d^{2n-1}f(a)}{dx^{2n-1}}\right], \tag{6.3}$$

where the numbers B_{2n} are the Bernouilli numbers.

The next observation is that, if the interval of integration is small enough, the series in the second term of equation 6.3 would yield a contribution negligible compared to that of the first term. Thus, we can write:

$$\int_a^b f(x)\,dx \approx \frac{b-a}{2}\left[f(a)+f(b)\right], \tag{6.4}$$

if $b-a$ is sufficiently small. The approximation of equation 6.4 represents the area of a trapeze whose summits are the circled points in Figure 6-2. Finally, one must remember the additive formula between integrals:

$$\int_a^b f(x)\,dx = \int_a^c f(x)\,dx + \int_c^b f(x)\,dx, \tag{6.5}$$

for any c. We shall use this property by chosing a c located between a and b.

The resulting strategy is a divide-and-conquer strategy. The integration interval is divided until one can be sure that the second term of equation 6.3 becomes indeed negligible. As one would like to re-use the points at which

the function has been evaluated during the course of the algorithm, the integration interval is halved at each iteration. The first few steps are outlined in figure 6-2. An estimation of the integral is obtained by summing the areas of the trapezes corresponding to each partition.

Let $x_0^{(n)}, \ldots, x_{2^n}^{(n)}$ be the partition of the interval at iteration n. Let $\epsilon^{(n)}$ be the length of each interval between these points. We thus have:

$$\begin{cases} \epsilon^{(n)} & = & \frac{b-a}{2^n} \\ x_0^{(n)} & = & a \\ x_i^{(n)} & = & a + i\epsilon^{(n)} \quad \text{for } i = 1, \ldots, 2^n. \end{cases} \tag{6.6}$$

The corresponding estimation for the integral is:

$$I^{(n)} = \epsilon^{(n)} \left[f(a) + f(b) + 2 \sum_{i=1}^{2^n - 1} f\left(x_i^{(n)}\right) \right]. \tag{6.7}$$

To compute the next estimation, it suffices to compute the value of the function at the even values of the partition because the odd values were already computed before. One can derive the following recurrence relation:

$$I^{(n+1)} = \frac{I^{(n)}}{2} + \epsilon^{(n)} \sum_{i=1}^{2^n - 1} f\left(x_{2i-1}^{(n)}\right), \tag{6.8}$$

with the initial condition:

$$I^{(0)} = \frac{b-a}{2} \left[f(a) + f(b) \right]. \tag{6.9}$$

Note that the sum on the right-hand side of equation 6.8 represents the sum of the function's values at the new points of the partition.

End game strategy

The final question is when should the algorithm be stopped? A real honest answer is we do not know. The magnitude of the series in equation 6.3 is difficult to estimate as the Bernouilli numbers become very large with increasing n. An experimental way is to watch for the change of the integral estimate In other words the absolute value of the last variation, $\left| I^{(n)} - I^{(n+1)} \right|$, is considered a good estimate of the precision. This kind of heuristic works for most functions.

At each iteration the number of function evaluation doubles. This means that the time spent in the algorithm grows exponentially with the number of iterations. Thus, the default maximum number of iteration must be kept quite low compared to that of the other methods.

In practice, however, trapeze integration converges quite slowly and should not be used. Why bother implementing it then? It turns out that the more elaborate methods, Simpson and Romberg integration, require the computation of the same sums needed by the trapeze integration. Thus, the trapeze integration is introduced to be the superclass of the other better integration methods.

One must keep in mind, however, that the magnitude of the series in equation 6.3 can become large for any function whose derivatives of high orders have singularities over the interval of integration. The convergence of the algorithm can be seriously compromised for such functions. This remark is true for the other algorithms described in this chapter. For example, none of the algorithms is able to give a precise estimate of the beta function using equation 2.29 with $x > 1$ and $y < 1$ (*c.f.* section 2.6) because, for these values, the derivatives of the function to integrate have a singularity at $t = 1$.

Another problem can come up if the function is nearly zeroes at regular intervals. For example, evaluating the integral of the function $f(x) = \frac{\sin(2^m x)}{x}$ from $-\pi$ to π for a moderate value of m. In this case, the terms $I^{(0)}$ to $I^{(m)}$ will have a null contribution. This would cause the algorithm to stop prematurely. Such special function behavior is of course quite rare. Nevertheless the reader must be aware of the limitations of the algorithm. This remark is valid for all algorithms exposed in this chapter.

Trapeze integration — Smalltalk implementation

The class implementing trapeze integration is a subclass of the functional iterator discussed in section 4.2. Two instance variables are needed to define the integration interval. Additional instance variables must keep track of the partition of the interval and of the successive estimations. Consequently, the class has the following instance variables.

- `from` contains the lower limit of the integration's interval, *i.e.* a,
- `to` contains the lower limit of the integration's interval, *i.e.* b,
- `step` contains the size of the interval's partition, *i.e.* $\epsilon^{(n)}$,
- `sum` contains the intermediate sums, *i.e.* $I^{(n)}$.

Although trapeze integration is not a practical algorithm, we give an example of implementation. The reason is that the public interface used by trapeze integration is the same for all integration classes.

The example shown in the next two sections is the integration of the inverse function. In mathematics the natural logarithm of x, $\ln x$, is defined as the integral from 1 to x of the inverse function. Of course, using numerical integration is a very inefficient way of computing a logarithm. This example, however, allows the reader to investigate the accuracy (since the exact solution is known) and performances of all algorithms presented in this chapter. The interested reader should try the example for various setting of the

desired precision and look at the number of iterations needed for a desired precision. She can also verify how accurate is the estimated precision.

Listing 6-3 shows the Smalltalk implementation of the trapeze integration method. In Smalltalk the code for the computation of the integral defining the natural logarithm is as follows:

```
| integrator ln2 ln3 |
integrator := PMTrapezeIntegrator
                function: [ :x | 1.0 / x ] from: 1 to: 2.
ln2 := integrator evaluate.
integrator from: 1 to: 3.
ln3 := integrator evaluate.
```

The line after the declaration creates a new instance of the class `PMTrapezeIntegrator` for the inverse function. The limits of the integration interval are set from 1 to 2 at creation time. The third line retrieves the value of the integral. The fourth line changes the integration interval and the last line retrieves the value of the integral over the new integration interval.

The class `PMTrapezeIntegrator` is a subclass of the class `PMAbstractFunctionIterator` defined in section 4.2.

In order to create an instance you need to use class method `function:from:to:` to define the fucntion and the integration interval.

The method `from:to:` allows changing the integration interval for a new computation with the same function.

Note that the initialization of the iterations (method `computeInitialValues`, *c.f.* section 4.1) also corresponds to the first iteration of the algorithm. The method `highOrderSum` computes the sum of the right-hand side of equation 6.8.

Listing 6-3 Smalltalk implementation of trapeze integration

```
PMFunctionalIterator subclass: #PMTrapezeIntegrator
   instanceVariableNames: 'from to sum step'
   classVariableNames: ''
   package: 'Math-DHB-Numerical'
```

```
PMTrapezeIntegrator class >> defaultMaximumIterations
    ^13
```

```
PMTrapezeIntegrator class >> function: aBlock from: aNumber1 to:
    aNumber2
    ^super new initialize: aBlock from: aNumber1 to: aNumber2
```

```
PMTrapezeIntegrator >> computeInitialValues
    step := to - from.
    sum := ((functionBlock value: from) + (functionBlock value: to))
    * step /2.
    result := sum.
```

```
PMTrapezeIntegrator >> evaluateIteration
    | oldResult |
    oldResult := result.
    result := self higherOrderSum.
    ^ self relativePrecision: ( result - oldResult) abs
```

```
PMTrapezeIntegrator >> from: aNumber1 to: aNumber2
    from := aNumber1.
    to := aNumber2
```

```
PMTrapezeIntegrator >> higherOrderSum
    | x newSum |
    x := step / 2 + from.
    newSum := 0.
    [ x < to ]
        whileTrue: [ newSum := ( functionBlock value: x) + newSum.
                     x := x + step.
                   ].
    sum := ( step * newSum + sum) / 2.
    step := step / 2.
    ^ sum
```

```
PMTrapezeIntegrator >> initialize: aBlock from: aNumber1 to: aNumber2
    functionBlock := aBlock.
    self from: aNumber1 to: aNumber2.
    ^ self
```

6.3 Simpson integration algorithm

Simpson integration algorithm consists in replacing the function to integrate by a second order Lagrange interpolation polynomial[**?**] (*c.f.* section 3.2). One can then carry the integral analytically. Let $f(x)$ be the function to integrate. For a given interval of integration, $[a, b]$, the function is evaluated at the extremities of the interval and at its middle point $c = \frac{a+b}{2}$. As defined in equation 3.1 the second order Lagrange interpolation polynomial is then given by:

$$P_2(x) = \frac{2}{(b-a)^2}\left[(x-b)(x-c)f(a) + (x-c)(x-a)f(b) + (x-a)(x-b)f(c)\right]. \tag{6.10}$$

The integral of the polynomial over the interpolation interval is given by:

$$\int_a^b P_2(x)\,dx = \frac{b-a}{6}\left[f(a) + f(b) + 4f(c)\right]. \tag{6.11}$$

As for trapeze integration, the interval is partitioned into small intervals. Let us assume that the interval has been divided into subintervals. By repeating

equation 6.11 over each subinterval, we obtain:

$$\int_a^b P_2(x)\,dx = \frac{\epsilon^{(n)}}{3}\left[f(a) + f(b) + 2\sum_{i=1}^{2^{n-1}} f\left(x_{2i-1}^{(n)}\right) + 4\sum_{i=0}^{2^{n-1}} f\left(x_{2i}^{(n)}\right)\right]. \tag{6.12}$$

Equation 6.12 uses the notations introduced in section 6.2. Except for the first iteration, the right-hand side of equation 6.12 can be computed from the quantities $I^{(n)}$ defined in equation 6.7. Thus, we have:

$$\int_a^b P_2(x)\,dx = \frac{1}{3}\left[4I^{(n)} - I^{(n-1)}\right] \quad \text{for } n > 1 \tag{6.13}$$

This can be checked by verifying that $I^{(n-1)}$ is equal to the first sum of equation 6.12 times and that $I^{(n)}$ is equal to the addition of the two sums of equation 6.12 times $\epsilon^{(n)}$. As advertised in section 6.2 we can re-use the major parts of the trapeze algorithm: computation of the sums and partition of the integration interval.

Like in the case of the trapeze algorithm, the precision of the algorithm is estimated by looking at the differences between the estimation obtained previously and the current estimation. At the first iteration only one function point is computed. This can cause the process to stop prematurely if the function is nearly zero at the middle of the integration interval. Thus, a protection must be built in to prevent the algorithm from stopping at the first iteration.

Simpson integration — Smalltalk implementation

The class implementing Simpson algorithm is a subclass of the class implementing trapeze integration. The method `evaluateIteration` is the only method needing change. The number of iterations is checked to prevent returning after the first iteration.

The public interface is the same as that of the superclass. Thus, all the examples shown in sections 6.2 can be used for Simpson algorithm by just changing the name of the class.

Listing 6-4 shows the complete implementation in Smalltalk.

The class `PMSimpsonIntegrator` is a subclass of the class `PMTrapezeIntegrator` defined in section 6.2.

Listing 6-4 Smalltalk implementation of the Simpson integration algorithm

```
PMTrapezeIntegrator subclass: #PMSimpsonIntegrator
   instanceVariableNames: ''
   classVariableNames: ''
   package: 'Math-DHB-Numerical'
```

```
PMSimpsonIntegrator >> evaluateIteration
| oldResult oldSum |
    iterations < 2
        ifTrue: [ self higherOrderSum.
                  ^ 1 ].
    oldResult := result.
    oldSum := sum.
    result := (self higherOrderSum * 4 - oldSum) / 3.
    ^self relativePrecision: ( result - oldResult) abs
```

6.4 Romberg integration algorithm

If one goes back to equation 6.3 one notices that the second term is of the order of the square of the integration interval. Romberg's algorithm uses this fact to postulate that $I^{(n)}$ is a smooth function of the square of the size of interval's partition $\epsilon^{(n)}$. Romberg's algorithm introduces the parameter k where $k-1$ is the degree of the interpolation's polynomial[2]. The result of the integral is estimated by extrapolating the series $I^{(n-k)}, \ldots, I^{(n)}$ at the value $\epsilon^{(n)} = 0$. Since we have:

$$\epsilon^{(n)} = \frac{\epsilon^{(n-1)}}{2}, \tag{6.14}$$

one only needs to interpolate over successive powers of $1/4$, starting with 1: $1, 1/4, 1/16, 1/256$, *etc*. In this case, extrapolation is safe because the value at which extrapolation is made is very close to the end of the interval defined by the sample points and actually becomes closer and closer with every iteration.

Thus, Romberg's algorithm requires at least k iterations. The good news is that this algorithm converges very quickly and, in general, only a few iterations are needed after the five initial ones. A polynomial of 4^{th} degree — that is $k = 5$ — is generally sufficient [**?**].

Extrapolation is performed using Neville's algorithm (*c.f.* section 3.4) because it computes an estimate of the error on the interpolated value. That error estimate can then be used as the estimate of the error on the final result.

If $k = 1$ Romberg's algorithm is equivalent to trapeze integration. If $k = 2$, the interpolation polynomial is given by:

$$P_1(x) = y_1 + \frac{x - x_1}{x_2 - x_1}(y_2 - y_1). \tag{6.15}$$

At the n^{th} iteration we have: $y_1 = I^{(n-1)}$, $y_2 = I^{(n)}$ and $x_2 = x_1/4$. Thus, the interpolated value at 0 is:

$$P_1(0) = I^{(n-1)} + \frac{4}{3}\left[I^{(n)} - I^{(n-1)}\right] = \frac{1}{3}\left[4I^{(n)} - I^{(n-1)}\right] \tag{6.16}$$

[2] In other words, k is the number of points over which the interpolation is performed (*c.f.* section 3.2).

Thus, for $k = 2$ Romberg's algorithm is equivalent to Simpson's algorithm. For higher order, however, Romberg's algorithm is much more efficient than Simpson method as soon as precision is required (*c.f.* a comparison of the result in section 6.6).

Using interpolation on the successive results of an iterative process to obtain the final result is a general procedure known as Richardson's deferred approach to the limit [**?**]. This technique can be used whenever the estimated error can be expressed as a function of a suitable parameter depending on the iterations. The choice of the parameter, however, is critical. For example, if one had interpolated over the size of the interval's partition instead of its square, the method would not converge as well[3].

Romberg integration — Smalltalk implementation

The class implementing Romberg's algorithm needs the following additional instance variables:

- `order` the order of the interpolation, *i.e.* k,
- `interpolaton` an instance of Neville's interpolation class,
- `points` an `OrderedCollection` containing the most recent sum estimates, *i.e.* $I^{(n-k)}, \ldots, I^{(n)}$.

The method `evaluateIteration` (*c.f.* section 4.1) contains the entire algorithm. At each iteration the collection of point receives a new point with an abscissa equal to the quarter of that of the last point and an ordinate equal to the next sum estimate . If not enough points are available, the method returns a precision such that the iterative process will continue. Otherwise, the extrapolation is performed. After the result of the extrapolation has been obtained the oldest point is removed. In other words, the collection of points is used as a last-in-last-out list with a constant number of elements equal to the order of the interpolation. Of the two values returned by Neville's interpolation (*c.f.* section 3.4), the interpolated value is stored in the result and the error estimate is returned as the precision for the other.

Listing 6-5 shows the Smalltalk implementation of Romberg's algorithm. The class `PMRombergIntegrator` is a subclass of the class `PMTrapezeIntegrator` defined in section 6.2.

The class method `defaultOrder` defines the default order to 5. This method is used in the method initialize so that each newly created instance is created with the default interpolation order. The method `order:` allows changing the default order if needed.

[3] It converges at the same speed as Simpson's algorithm. This can be verified by running the comparison programs after changing the factor 0.25 used to compute the abscissa of the next point into 0.5.

The sample points defining the interpolation are stored in an `OrderedCollection`. This collection is created in the method `computeInitialValues`. Since the number of points will never exceed the order of the interpolation the maximum size is preset when the collection is created. The method `computeInitialValues` also creates the object in charge of performing the interpolation and it stores the first point in the collection of sample points.

Listing 6-5 Smalltalk implementation of Romberg integration

```
PMTrapezeIntegrator subclass: #PMRombergIntegrator
   instanceVariableNames: 'order points interpolator'
   classVariableNames: ''
   package: 'Math-DHB-Numerical'
```

```
PMRombergIntegrator class >> defaultOrder
    ^5
```

```
PMRombergIntegrator >> computeInitialValues
    super computeInitialValues.
    points := OrderedCollection new: order.
    interpolator := PMNevilleInterpolator points: points.
    points add: 1 @ sum
```

```
PMRombergIntegrator >> evaluateIteration
    | interpolation |
    points addLast: (points last x * 0.25) @ self higherOrderSum.
    points size < order
        ifTrue: [ ^1].
    interpolation := interpolator valueAndError: 0.
    points removeFirst.
    result := interpolation at: 1.
    ^self relativePrecision: ( interpolation at: 2) abs
```

```
PMRombergIntegrator >> initialize
    order := self class defaultOrder.
    ^ super initialize
```

```
PMRombergIntegrator >> order: anInteger
    anInteger < 2
        ifTrue: [
           self error: 'Order for Romberg integration must be
                                                larger than 1'].
    order := anInteger
```

6.5 Evaluation of open integrals

An open integral is an integral for which the function to integrate cannot be evaluated at the boundaries of the integration interval. This is the case when one of the limit is infinite or when the function to integrate has a singularity at one of the limits. If the function to integrate has one or more singularity in

the middle of the integration interval, the case can be reduced to that of having a singularity at the limits using the additive formula between integrals 6.5. Generalization of the trapeze algorithm and the corresponding adaptation of Romberg's algorithm can be found in [**?**].

Bag of tricks

My experience is that using a suitable change of variable can often remove the problem. In particular, integrals whose integration interval extends to infinity can be rewritten as integrals over a finite interval. We give a few examples below.

For an integral starting from minus infinity, a change of variable $t = \frac{1}{x}$ can be used as follows:

$$\int_{-\infty}^{a} f(x)\,dx = \int_{\frac{1}{a}}^{0} f\left(\frac{1}{t}\right)\frac{dt}{t^2} \quad \text{for } a < 0. \tag{6.17}$$

For such integral to be defined, the function must vanish at minus infinity faster than x^2. This means that:

$$\lim_{t\to 0} \frac{1}{t^2} f\left(\frac{1}{t}\right) = 0. \tag{6.18}$$

If $a > 0$, the integration must be evaluated in two steps, for example one over the interval $]-\infty, -1]$ using the change of variable of equation 6.17 and one over the interval $[-1, a]$ using the original function.

For integral ending at positive infinity the same change of variable can also be made. However, if the interval of integration is positive, the change of variable $t = e^{-x}$ can be more efficient. In this case, one makes the following transformation:

$$\int_{a}^{+\infty} f(x)\,dx = \int_{0}^{e^{-a}} f(-\ln t)\,\frac{dt}{t} \quad \text{for } a > 0. \tag{6.19}$$

For this integral to be defined one must have:

$$\lim_{t\to 0} \frac{1}{t} f(\ln t) = 0. \tag{6.20}$$

By breaking up the interval of integration is several pieces one can chose a change of variable most appropriate for each piece.

6.6 Which method to chose?

An example comparing the results of the three algorithms is given in section 6.6 for Smalltalk. The function to integrate is the inverse function in both

cases. The integration interval is from 1 to 2 so that the value of the result is known, namely $\ln 2$. Integration is carried for various values of the desired precision. The reader can then compare the attained precision (both predicted and real) and the number of iterations[4] required for each algorithm. Let us recall that the number of required function evaluations grows exponentially with the number of iterations.

The results clearly show that the trapeze algorithm is ruled out as a practical method. As advertised in section 6.2 it is not converging quickly toward a solution.

Romberg's algorithm is the clear winner. At given precision, it requires the least number of iterations. This is the algorithm of choice in most cases.

Simspon's algorithm may be useful if the required precision is not too high and if the time to evaluate the function is small compared to the interpolation. In such cases Simspon's algorithm can be faster than Romberg's algorithm.

Section 6.6 give some sample code the reader can use to investigate the various integration algorithms. The results of the code execution are shown in table 6-6. The columns of this table are:

- ϵmax the desired precision,
- n the number of required iterations; let us recall that the corresponding number of function's evaluations is 2^{n+1},
- $\tilde{\epsilon}$ the estimated precision of the result,
- ϵ the effective precision of the result, that is the absolute value of the difference between the result of the integration algorithm and the true result.

Table 6-6 Comparison between integration algorithms

	Trapeze algorithm			Simpson algorithm			Romberg algorithm		
ϵmax	n	$\tilde{\epsilon}$	ϵ	n	$\tilde{\epsilon}$	ϵ	n	$\tilde{\epsilon}$	ϵ
10^{-5}	8	$4.1 \cdot 10^{-6}$	$9.5 \cdot 10^{-7}$	4	$9.9 \cdot 10^{-6}$	$4.7 \cdot 10^{-7}$	4	$1.7 \cdot 10^{-9}$	$1.4 \cdot 10$
10^{-7}	11	$6.4 \cdot 10^{-8}$	$1.5 \cdot 10^{-8}$	6	$4.0 \cdot 10^{-8}$	$1.9 \cdot 10^{-9}$	4	$1.7 \cdot 10^{-9}$	$1.4 \cdot 10$
10^{-9}	15	$2.5 \cdot 10^{-10}$	$5.8 \cdot 10^{-11}$	8	$1.5 \cdot 10^{-10}$	$7.3 \cdot 10^{-12}$	5	$1.4 \cdot 10^{-11}$	$3.7 \cdot 10$
10^{-11}	18	$3.9 \cdot 10^{-12}$	$9.0 \cdot 10^{-13}$	9	$9.8 \cdot 10^{-12}$	$5.7 \cdot 10^{-13}$	6	$7.6 \cdot 10^{-14}$	$5.7 \cdot 10$
10^{-13}	21	$4.8 \cdot 10^{-14}$	$2.8 \cdot 10^{-14}$	11	$3.8 \cdot 10^{-14}$	$1.9 \cdot 10^{-15}$	6	$7.6 \cdot 10^{-14}$	$5.7 \cdot 10$

Smalltalk comparison

The script of Listing 6-7 can be executed as such in a playground window.

[4]Note that the number of iterations printed in the examples in one less than the real number of iterations because the first iteration is performed in the set-up phase of the iterative process.

The function to integrate is specified as a block closure as discussed in section 2.2.

Listing 6-7 Smalltalk comparison script for integration algorithms

```
| a b  integrators |
a := 1.0.
b := 2.0.
integrators := Array with: (
            (PMTrapezeIntegrator function: [ :x | 1.0 / x]) from: a
    to: b)
            with: (
            (PMSimpsonIntegrator function: [ :x | 1.0 / x]) from: a
    to: b)
            with: (
            (PMRombergIntegrator function: [ :x | 1.0 / x]) from: a
    to: b).
#(1.0e-5 1.0e-7 1.0e-9 1.0e-11 1.0e-13) do:
[ :precision |
   Transcript cr; cr; nextPutAll: '===> Precision: '.
   precision printOn: Transcript.
   integrators do:
      [ :integrator |
      Transcript cr; nextPutAll: '*****   ', integrator class
    name,':'; cr.
      integrator desiredPrecision: precision.
      Transcript nextPutAll: 'Integral of 1/x from '.
      a printOn: Transcript.
      Transcript nextPutAll: ' to '.
      b printOn: Transcript.
      Transcript nextPutAll: ' = '.
      integrator evaluate printOn: Transcript.
      Transcript nextPutAll: ' +- '.
      integrator precision printOn: Transcript.
      Transcript cr; nextPutAll: ' ( '.
      integrator iterations printOn: Transcript.
      Transcript nextPutAll: ' iterations, true error = '.
      ( integrator result - 2 ln) printOn: Transcript.
      Transcript nextPutAll: ')'; cr].
   Transcript flush.
]
```

CHAPTER 7

Series

*O*n ne peut pas partir de l'infini, on peut y aller.[1]
Jules Lachelier

Whole families of functions are defined with infinite series expansion or a continued fraction. Before the advent of mechanical calculators, a person could earn a Ph.D. in mathematics by publishing tabulated values of a function evaluated by its series expansion or continued fraction. Some people developed a talent to perform such tasks.

Some reported stories make the task of evaluating series sound like a real business. A German nobleman discovered that one of its peasants had a talent for numbers. He then housed him in his mansion and put him to work on the calculation of a table of logarithms. The table was published under the nobleman's name[**?**].

Nowadays we do not need to look for some talented peasant, but we still publish the result of computations made by other than ourselves. Overall computers are better treated than peasants were, though. . .

7.1 Introduction

It will not come as a surprise to the reader that the computation of infinite series is made on a computer by computing a sufficient but finite number of terms. The same is true for continued fractions. Thus, the computation of infinite series and continued fractions uses the iterative process framework described in chapter 4. In this case the iteration consists of computing successive terms.

[1] One cannot start at infinity; one can reach it, however.

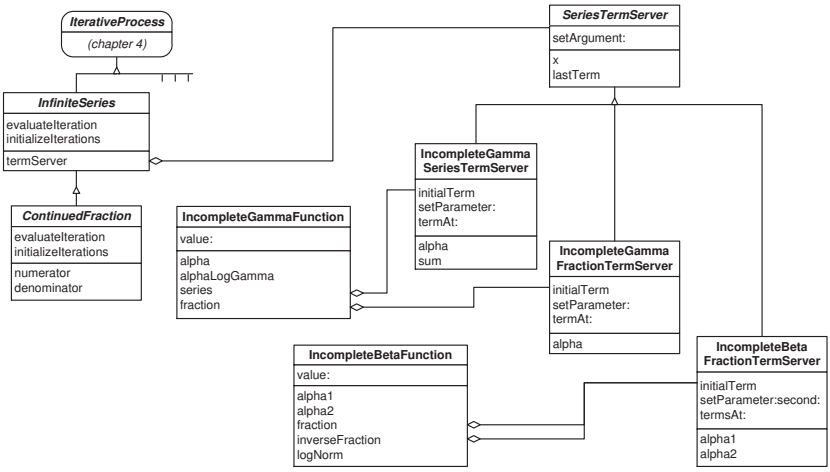


Figure 7-1 Smalltalk class diagram for infinite series and continued fractions

The present chapter begins by exposing a general framework on how to compute infinite series and continued fractions. Then, we show two examples of application of this framework by implementing two functions, which are very important to compute probabilities: the incomplete gamma function and the incomplete beta function.

Figure 7-1 shows the class diagram of the Smalltalk implementation.

The Smalltalk implementation uses two general-purpose classes to implement an infinite series and a continued fraction respectively. Each class then use a Strategy pattern class [**?**] to compute each term of the expansion.

7.2 Infinite series

Many functions are defined with an infinite series, that is a sum of an infinite number of terms. The most well known example is the series for the exponential function:

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}. \tag{7.1}$$

For such a series to be defined, the terms of the series must become very small as the index increases. If that is the case an infinite series may be used to evaluate a function, for which no closed expression exists. For this to be practical, however, the series should converge quickly so that only a few terms are needed. For example, computing the exponential of 6 to the precision of an IEEE 32 bit floating number requires nearly 40 terms. This is clearly not an efficient way to compute the exponential.

Discussing the convergence of a series is outside of the scope of this book. Let us just state that in general numerical convergence of a series is much harder to achieve than mathematical convergence. In other words the fact that a series is defined mathematically does not ensure that it can be evaluated numerically.

A special remark pertains to alternating series. In an alternating series the signs of consecutive terms are opposite. Trigonometric functions have such a series expansion. Alternating series have very bad numerical convergence properties: if the terms are large rounding errors might suppress the convergence altogether. If one cannot compute the function in another way it is best to compute the terms of an alternating series in pairs to avoid rounding errors.

In practice, a series must be tested for quick numerical convergence prior to its implementation. As for rounding errors the safest way is to do this experimentally, that is, print out the terms of the series for a few representative[2] values of the variable. In the rest of this chapter we shall assume that this essential step has been made.

To evaluate an infinite series, one carries the summation until the last added term becomes smaller than the desired precision. This kind of logic is quite similar to that of an iterative process. Thus, the object used to compute an infinite series belongs to a subclass of the iterative process classe discussed in chapter 4.

Infinite series — Smalltalk implementation

Listing 7-2 shows a general Smalltalk implementation of a class evaluating an infinite series. The class being abstract, we do not give examples here. Concrete examples are given in section 7.4.

The Smalltalk implementation uses a Strategy pattern. The class `PMInfiniteSeries` is a subclass of the class `IterativeProcess`, discussed in section 4.1. This class does not implement the algorithm needed to compute the terms of the series directly. It delegates this responsibility to an object stored in the instance variable `termServer`. Two hook methods, `initialTerm` and `termAt:` are used to obtain the terms of the series from the term server object.

The method `evaluateIteration` uses the method `precisionOf:relativeTo:` to return a relative precision as discussed in section 4.2.

To implement a specific series, an object of the class `PMInfiniteSeries` is instantiated with a specific term server. A concrete example will be shown in section 7.4.

Because of its generic nature, the class `PMInfiniteSeries` does not implement the function behavior described in section 2.2 (method `value:`). It is

[2] By representative, I mean either values, which are covering the domain over which the function will be evaluated, or values, which are suspected to give convergence problems.

the responsibility of each object combining an infinite series with a specific term server to implement the function behavior. An example is given in section 7.4.

Listing 7-2 Smalltalk implementation of an infinite series

```
PMIterativeProcess subclass: #PMInfiniteSeries
   instanceVariableNames: 'termServer'
   classVariableNames: ''
   package: 'Math-Series'
```

```
PMInfiniteSeries class >> server: aTermServer
    ^self new initialize: aTermServer
```

```
PMInfiniteSeries >> evaluateIteration
    | delta |
    delta := termServer termAt: iterations.
    result := result + delta.
    ^ self precisionOf: delta abs relativeTo: result abs
```

```
PMInfiniteSeries >> initialize: aTermServer
    termServer := aTermServer.
    ^ self
```

```
PMInfiniteSeries >> initializeIterations
    result := termServer initialTerm
```

The computation of the terms of the series is delegated to an object instantiated from a server class. The abstract server class is called `PMInfiniteSeriesTermServer`. It is responsible to compute the terms at each iteration. This class receives the argument of the function defined by the series, which is kept in the instance variable `x`. The instance variable `lastTerm` is provided to keep the last computed term since the next term can often be computed from the previous one. The code of this abstract class is shown in Listing 44.

Listing 7-3 Smalltalk implementation of a term server

```
Object subclass: #PMSeriesTermServer
   instanceVariableNames: 'x lastTerm'
   classVariableNames: ''
   package: 'Math-DHB-Numerical'
```

```
PMSeriesTermServer >> setArgument: aNumber
    x := aNumber asFloat
```

7.3 Continued fractions

A continued fraction is an infinite series of cascading fractions of the following form:

$$f(x) = b_0 + \cfrac{a_1}{b_1 + \cfrac{a_2}{b_2 + \cfrac{a_3}{b_3 + \cfrac{a_4}{b_4 + \cdots}}}} \tag{7.2}$$

In general, both sets of coefficients $a_0, \ldots$ and $b_0, \ldots$ depend on the function's argument x. This dependence in implicit in equation 7.2 to keep the notation simple. Since the above expression is quite awkward to read - besides being a printer's nightmare - one usually uses a linear notation as follows [**?**], [**?**]:

$$f(x) = b_0 + \frac{a_1}{b_1+}\frac{a_2}{b_2+}\frac{a_3}{b_3+}\frac{a_4}{b_4+}\cdots \tag{7.3}$$

The problem in evaluating such a fraction is that, *a* priori, one must begin the evaluation from the last term. Fortunately, methods allowing the evaluation from the beginning of the fractions have been around since the seventeen's century. A detailed discussion of several methods is given in [**?**]. In this book, we shall only discuss the modified Lentz' method which has the advantage to work for a large class of fractions.

Implementing the other methods discussed in [**?**] is left as an exercise to the reader. The corresponding classes can be subclassed from the classes found in this chapter.

In 1976, Lentz proposed the following two auxiliary series:

$$\begin{cases} C_0 &= b_0, \\ D_0 &= 0, \\ C_n &= \frac{a_n}{C_{n-1}} + b_n \quad \text{for } n > 0, \\ D_n &= \frac{1}{a_n D_{n-1} + b_n} \quad \text{for } n > 0. \end{cases} \tag{7.4}$$

These two series are used to construct the series:

$$\begin{cases} f_0 &= C_0, \\ f_n &= f_{n-1} C_n D_n. \end{cases} \tag{7.5}$$

One can prove by induction that this series converges toward the continued fraction as n gets large.

In general continued fractions have excellent convergence properties. Some care, however, must be given when one of the auxiliary terms C_n or $1/D_n$ become nearly zero[3]. To avoid rounding errors, Thompson and Barnett, in 1986, proposed a modification of the Lentz method in which any value of the

[3] That is, a value which is zero within the precision of the numerical representation.

coefficients smaller than a small floor value is adjusted to the floor value [?]. The floor value is chosen to be the machine precision of the floating-point representation (instance variable `smallNumber` described in section 1.4).

In terms of architecture, the implementation of a continued fraction is similar to that of the infinite series.

Continued fractions — Smalltalk implementation

Listing 7-4 shows the implementation of a continued fraction in Smalltalk.

The class `PMContinuedFraction` is built as a subclass of the class `PMInfiniteSeries`. Thus, it uses also the Strategy pattern.

The method `limitedSmallValue:` implements the prescription of Thompson and Barnett.

Listing 7-4 Smalltalk implementation of a continued fraction

```
PMInfiniteSeries subclass: #PMContinuedFraction
   instanceVariableNames: 'numerator denominator'
   classVariableNames: ''
   package: 'Math-Series'
```

```
PMContinuedFraction >> evaluateIteration
    | terms delta |
    terms := termServer termsAt: iterations.
    denominator := 1 / (self limitedSmallValue: ((terms at: 1) *
    denominator + (terms at: 2))).
    numerator := self limitedSmallValue: ((terms at: 1) / numerator
    + (terms at: 2)).
    delta := numerator * denominator.
    result := result * delta.
    ^ (delta - 1) abs
```

```
PMContinuedFraction >> initializeIterations
    numerator := self limitedSmallValue: termServer initialTerm.
    denominator := 0.
    result := numerator
```

```
limitedSmallValue: aNumber
    ^aNumber abs < PMFloatingPointMachine new smallNumber
            ifTrue: [ PMFloatingPointMachine new smallNumber ]
            ifFalse: [ aNumber ]
```

7.4 Incomplete Gamma function

The incomplete gamma function is the integral of a gamma distribution. It 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 gamma distribution. In particular, the incomplete gamma function is used

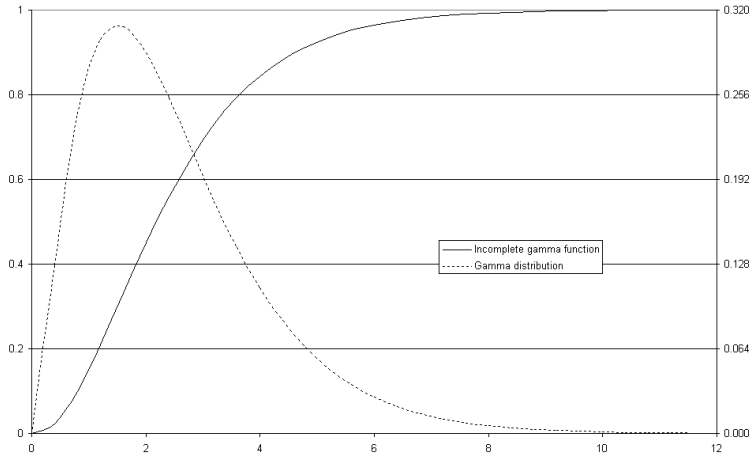


Figure 7-5 The incomplete gamma function and the gamma distribution

to compute the confidence level of χ^2 values when assessing the validity of a parametric fit. Several examples of use of this function will be introduced in chapters 9 and 10.

Figure 7-5 shows the incomplete gamma function (solid line) and its corresponding probability density function (dotted line) for $\alpha = 2.5$.

The gamma distribution is discussed in section 9.7. The χ^2 confidence level is discussed in section 10.3. General χ^2 fits are discussed in section 10.9.

Mathematical definitions

The incomplete gamma function is defined by the following integral:

$$\Gamma(x, \alpha) = \frac{1}{\Gamma(\alpha)} \int_0^x t^{\alpha-1} e^{-t} dt. \tag{7.6}$$

Thus, the value of the incomplete gamma function lies between 0 and 1. The function has one parameter α. The incomplete gamma function is the distribution function of a gamma probability density function with parameters α and 1 (*c.f.* section 9.7 for a description of the gamma distribution and its parameters). This integral can be expressed as the following infinite series [**?**]:

$$\Gamma(x, \alpha) = \frac{e^{-x} x^{\alpha}}{\Gamma(\alpha)} \sum_{n=0}^{\infty} \frac{\Gamma(\alpha)}{\Gamma(\alpha + 1 + n)} x^n. \tag{7.7}$$

Written in this form we can see that each term of the series can be computed from the previous one. Using the recurrence formula for the gamma function

— equation 2.21 in section 2.5 — we have:

$$\begin{cases} a_0 &= \frac{1}{\alpha}, \\ a_n &= \frac{x}{\alpha+1+n} a_{n-1}. \end{cases} \tag{7.8}$$

The series in equation 7.7 converges well for $x < \alpha + 1$.

The incomplete gamma function can also be written as [**?**]:

$$\Gamma(x, \alpha) = \frac{e^{-x} x^{\alpha}}{\Gamma(\alpha)} \frac{1}{F(x - \alpha + 1, \alpha)}, \tag{7.9}$$

where $F(x, \alpha)$ is the continued fraction:

$$F(x, \alpha) = x + \frac{1(\alpha - 1)}{x + 2+} \frac{2(\alpha - 2)}{x + 4+} \frac{3(\alpha - 3)}{x + 6+} \ldots \tag{7.10}$$

Using the notation introduced in equation 7.3 in section 7.3 the terms of the continued fraction are given by the following expressions:

$$\begin{cases} b_n &= x - \alpha + 2n \quad \text{for } n = 0, 1, 2, \ldots \\ a_n &= n(\alpha - n) \quad \text{for } n = 1, 2, \ldots \end{cases} \tag{7.11}$$

It turns out that the continued fraction in equation 7.9 converges for $x > \alpha + 1$ [**?**], that is, exactly where the series expansion of equation 7.7 did not converge very well. Thus, the incomplete gamma function can be computed using one of the two methods depending on the range of the argument.

The reader will notice that equations 7.7 and 7.9 have a common factor. The denominator of that factor can be evaluated in advance in logarithmic form to avoid floating-point overflow (*c.f.* discussion in section 2.5). For each function evaluation the entire factor is computed in logarithmic form to reduce rounding errors. Then it is combined with the value of the series or the continued fraction to compute the final result.

To avoid a floating-point error when evaluating the common factor, the value of the incomplete gamma function at $x = 0$ — which is of course 0 — must be returned separately.

Incomplete Gamma function — Smalltalk implementation

Three classes are needed to implement the incomplete gamma function in Smalltalk. The class `PMIncompleteGamaFunction` is in charge of computing the function itself. This is the object, which responds to the method `value:` to provide a function-like behavior to the object. It is shown in Listing 7-6 and has the following instance variables.

- `alpha` contains the function's parameter, *i.e.* α,
- `alphaLogGamma` used to cache the value of $\Gamma(\alpha)$ for efficiency purposes,

- `series` contains the infinite series associated to the function,
- `fraction` contains the continued fraction associated to the function.

The instance variables `series` and `fraction` are assigned using lazy initialization.

Depending on the range of the argument, the class delegates the rest of the computing to either a series or a continued fraction. In each case, a term server class provides the computation of the terms. They are shown in listings 7-7 and 7-8.

Listing 7-6 Smalltalk implementation of the incomplete gamma function

```
Object subclass: #PMIncompleteGammaFunction
   instanceVariableNames: 'alpha alphaLogGamma series fraction'
   classVariableNames: ''
   package: 'Math-DistributionGamma'
```

```
PMIncompleteGammaFunction class >> shape: aNumber
   ^self new initialize: aNumber
```

```
PMIncompleteGammaFunction >> evaluateFraction: aNumber
    fraction isNil
        ifTrue:
            [fraction := PMIncompleteGammaFractionTermServer new.
            fraction setParameter: alpha].
    fraction setArgument: aNumber.
    ^(PMContinuedFraction server: fraction)
        desiredPrecision: PMFloatingPointMachine new
    defaultNumericalPrecision;
        evaluate
```

```
PMIncompleteGammaFunction >> evaluateSeries: aNumber
    series isNil
        ifTrue: [ series := PMIncompleteGammaSeriesTermServer new.
                  series setParameter: alpha.
                ].
    series setArgument: aNumber.
    ^ (PMInfiniteSeries server: series)
        desiredPrecision: PMFloatingPointMachine new
    defaultNumericalPrecision;
        evaluate
```

```
PMIncompleteGammaFunction >> initialize: aNumber
    alpha := aNumber asFloat.
    alphaLogGamma := alpha logGamma.
    ^ self
```

```
PMIncompleteGammaFunction >> value: aNumber
    | x norm |
    aNumber = 0
        ifTrue: [ ^0].
    x := aNumber asFloat.
```

```
    norm := [ ( x ln * alpha - x - alphaLogGamma) exp] when: ExAll
                                do: [ :signal | signal exitWith: nil
    ].
    norm isNil
        ifTrue: [ ^1].
    ^x - 1 < alpha
        ifTrue: [ ( self evaluateSeries: x) * norm ]
        ifFalse: [ 1 - ( norm / ( self evaluateFraction: x)) ]
```

Listing 7-7 shows the implementation of the term server for the series expansion. It needs two instance variables: one to store the parameter α; one to store the sum accumulated in the denominator of equation 7.8. The two lines of equation 7.8 are implemented respectively by the methods `initialTerm` (for $n = 0$) and `termAt:` (for $n \geq 1$).

Listing 7-7 Smalltalk implementation of the series term server for the incomplete gamma function

```
PMSeriesTermServer subclass: #PMIncompleteGammaSeriesTermServer
   instanceVariableNames: 'alpha sum'
   classVariableNames: ''
   package: 'Math-DHB-Numerical'
```

```
PMIncompleteGammaSeriesTermServer >> initialTerm
  lastTerm := 1 / alpha.
    sum := alpha.
    ^ lastTerm
```

```
PMIncompleteGammaSeriesTermServer >> setParameter: aNumber
    alpha := aNumber asFloat
```

```
PMIncompleteGammaSeriesTermServer >> termAt: anInteger
  sum := sum + 1.
    lastTerm := lastTerm * x / sum.
    ^ lastTerm
```

Listing 7-8 shows the implementation of the term server for the continued fraction. It needs one instance variable to store the parameter α. Equation 7.11 is implemented by the methods `initialTerm` (for $n = 0$) and `termsAt:` (for $n \geq 1$).

Listing 7-8 Smalltalk implementation of the fraction term server for the incomplete gamma function

```
PMSeriesTermServer subclass: #PMIncompleteGammaFractionTermServer
   instanceVariableNames: 'alpha'
   classVariableNames: ''
   package: 'Math-DHB-Numerical'
```

```
PMIncompleteGammaFractionTermServer >> initialTerm
    lastTerm := x - alpha + 1.
    ^ lastTerm
```

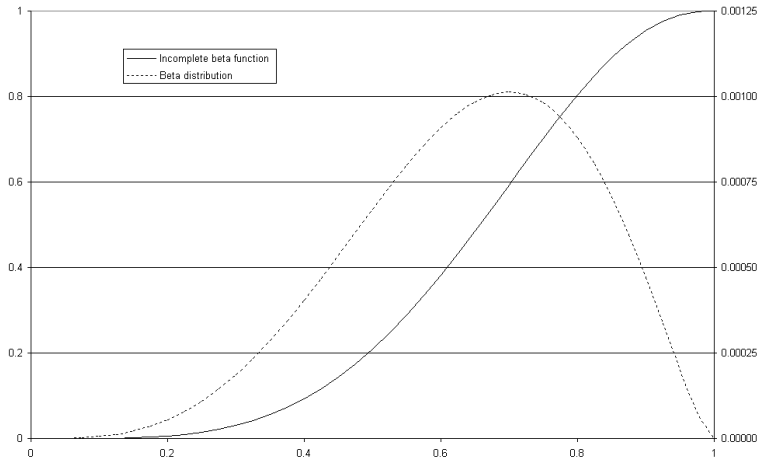


Figure 7-9 The incomplete beta function and the beta distribution

```
PMIncompleteGammaFractionTermServer >> setParameter: aNumber
    alpha := aNumber asFloat
```

```
PMIncompleteGammaFractionTermServer >> termsAt: anInteger
    lastTerm := lastTerm + 2.
    ^ Array with: (alpha - anInteger) * anInteger with: lastTerm
```

An example of use of the incomplete gamma function can be found in section 9.7.

7.5 Incomplete Beta function

The incomplete beta function is the integral of a beta distribution. It used in statistics to evaluate the probability of finding a measurement larger than a given value when the measurements are distributed according to a beta distribution. It is also used to compute the confidence level of the Student distribution (t-test) and of the Fisher-Snedecor distribution (F-test). The beta distribution is discussed in section **??**. The t-test is discussed in section 10.2. The F-test is discussed in section 10.1.

Figure 7-9 shows the incomplete beta function (solid line) and its corresponding probability density function (dotted line) for $\alpha_1 = 4.5$ and $\alpha_2 = 2.5$.

Mathematical definitions

The incomplete beta function is defined over the interval $[0, 1]$ by the following integral:

$$B(x; \alpha_1, \alpha_2) = \frac{1}{B(\alpha_1, \alpha_2)} \int_0^x t^{\alpha_1 - 1} (1-t)^{\alpha_2 - 1} \, dt, \tag{7.12}$$

where $B(\alpha_1, \alpha_2)$ is the beta function defined in section 2.6. The function has two parameters α_1 and α_2. By definition, the value of the incomplete beta function is comprised between 0 and 1.

None of the series expansions of this integral have good numerical convergence. There is, however, a continued fraction development which converges over a sufficient range [**?**]:

$$B(x; \alpha_1, \alpha_2) = \frac{x^{\alpha_1 - 1} (1-x)^{\alpha_2 - 1}}{\alpha_1 B(\alpha_1, \alpha_2)} \frac{1}{F(x; \alpha_1, \alpha_2)}, \tag{7.13}$$

where

$$F(x; \alpha_1, \alpha_2) = 1 + \frac{a_1}{1+} \frac{a_2}{1+} \frac{a_3}{1+} \cdots \tag{7.14}$$

Using the notation introduced in section 7.3 we have:

$$\begin{cases} b_n & = & 1 \quad \text{for } n = 0, 1, 2, \ldots \\ a_{2n} & = & \dfrac{n(\alpha_2 - n)x}{(\alpha_1 + 2n)(\alpha_1 + 2n - 1)} \quad \text{for } n = 1, 2, \ldots \\ a_{2n+1} & = & \dfrac{(\alpha_1 + n)(\alpha_1 + \alpha_2 + n)x}{(\alpha_1 + 2n)(\alpha_1 + 2n - 1)} \quad \text{for } n = 1, 2, \ldots \end{cases} \tag{7.15}$$

The continued fraction in equation 7.13 converges rapidly for $x > \frac{\alpha_1 + 1}{\alpha_1 + \alpha_2 + 2}$ [**?**]. To compute the incomplete beta function over the complementary range, one uses the following symmetry property of the function:

$$B(x; \alpha_1, \alpha_2) = 1 - B(1 - x; \alpha_2, \alpha_1) \tag{7.16}$$

Since $1 - x < \frac{\alpha_2 + 1}{\alpha_1 + \alpha_2 + 2}$ if $x < \frac{\alpha_1 + 1}{\alpha_1 + \alpha_2 + 2}$, we can now compute the function over the entire range.

To avoid a floating-point error when evaluating the leading factor of equation 7.13, the values of the incomplete beta function at $x = 0$ — which is 0 — and at $x = 1$ — which is 1 — must be returned separately.

Incomplete Beta function — Smalltalk implementation

Listing 7-10 shows the implementation of the incomplete beta function in Smalltalk.

Two classes are needed to implement the incomplete beta function. The class `PMIncompleteBetaFunction` is in charge of computing the function itself. This class has the following instance variables.

- `alpha1` contains the first function's parameter, i.e. α_1,
- `alpha2` contains the second function's parameter, i.e. α_2,
- `logNorm` used to cache the value of $\ln B(\alpha_1, \alpha_2)$ for efficiency purposes,
- `fraction` contains the continued fraction associated to the function $B(x; \alpha_1, \alpha_2)$,
- `inverseFraction` contains the continued fraction associated to the function $B(1 - x; \alpha_2, \alpha_1)$.

Depending on the range of the argument, the class delegates the rest of the computing to a continued fraction using the original parameters or the reversed parameters if the symmetry relation must be used. A term server class allows the computing of the terms. Its code is shown in listing 7-11. The two instance variables - `fraction` and `inverseFraction` -, contain an instance of the term server, one for each permutation of the parameters, thus preventing the unnecessary creation of new instances of the term server at each evaluation. These instance variables are assigned using lazy initialization.

Listing 7-10 Smalltalk implementation of the incomplete beta function

```
Object subclass: #PMIncompleteBetaFunction
   instanceVariableNames: 'alpha1 alpha2 fraction inverseFraction
    logNorm'
   classVariableNames: ''
   package: 'Math-DistributionBeta'
```

```
PMIncompleteBetaFunction class >> shape: aNumber1 shape: aNumber2
   ^self new initialize: aNumber1 shape: aNumber2
```

```
PMIncompleteBetaFunction >>evaluateFraction: aNumber
    fraction isNil
        ifTrue:
            [ fraction := PMIncompleteBetaFractionTermServer new.
            fraction setParameter: alpha1 second: alpha2 ].
    fraction setArgument: aNumber.
    ^(PMContinuedFraction server: fraction)
        desiredPrecision: PMFloatingPointMachine new
    defaultNumericalPrecision;
        evaluate
```

```
PMIncompleteBetaFunction >> evaluateInverseFraction: aNumber
    inverseFraction isNil
        ifTrue:
            [ inverseFraction := PMIncompleteBetaFractionTermServer
    new.
            inverseFraction setParameter: alpha2 second: alpha1 ].
    inverseFraction setArgument: (1 - aNumber).
    ^(PMContinuedFraction server: inverseFraction)
        desiredPrecision: PMFloatingPointMachine new
    defaultNumericalPrecision;
```

```
        evaluate
```

```
PMIncompleteBetaFunction >> initialize: aNumber1 shape: aNumber2
    alpha1 := aNumber1.
    alpha2 := aNumber2.
    logNorm := ( alpha1 + alpha2) logGamma - alpha1 logGamma -
    alpha2 logGamma.
    ^ self
```

```
PMIncompleteBetaFunction >> value: aNumber
    | norm |
    aNumber = 0
        ifTrue: [ ^ 0 ].
    aNumber = 1
        ifTrue: [ ^ 1 ].
    norm :=  (aNumber ln * alpha1 + ((1 - aNumber) ln * alpha2) +
    logNorm) exp.
    ^ (alpha1 + alpha2 + 2) * aNumber < (alpha1 + 1)
        ifTrue: [ norm / ( ( self evaluateFraction: aNumber) *
    alpha1)]
        ifFalse: [ 1 - ( norm / ( ( self evaluateInverseFraction:
    aNumber) * alpha2)) ]
```

Listing 7-11 shows the implementation of the term server. It needs two instance variables to store the parameters α_1 and α_2. Equation 7.15 is implemented by the methods `initialTerm` (for $n = 0$) and `termsAt:` (for $n \geq 1$).

Listing 7-11 Smalltalk implementation of the term server for the incomplete beta function

```
PMSeriesTermServer subclass: #PMIncompleteBetaFractionTermServer
   instanceVariableNames: 'alpha1 alpha2'
   classVariableNames: ''
   package: 'Math-DHB-Numerical'
```

```
PMIncompleteBetaFractionTermServer >> initialTerm
    ^ 1
```

```
PMIncompleteBetaFractionTermServer >> setParameter: aNumber1 second:
    aNumber2
    alpha1 := aNumber1.
    alpha2 := aNumber2
```

PMIncompleteBetaFractionTermServer » termsAt: anInteger

```
    | n n2 |
    n := anInteger // 2.
    n2 := 2 * n.
    ^Array with: ( n2 < anInteger
        ifTrue: [ x negated * (alpha1 + n) * (alpha1 + alpha2 + n)
                                / ((alpha1 + n2) * (alpha1 + 1 +
    n2))]
        ifFalse: [x * n * (alpha2 - n) / ((alpha1 + n2) * (alpha1 -
    1 + n2)) ])
```

```
            with: 1
```

An example of use of the incomplete beta function can be found in sections 10.1 and 10.2.

CHAPTER 8

Linear algebra

On ne trouve pas l'espace, il faut toujours le construire.[1]
Gaston Bachelard

Linear algebra concerns itself with the manipulation of vectors and matrices. The concepts of linear algebra are not difficult and linear algebra is usually taught in the first year of university. Solving systems of linear equations are even taught in high school. Of course, one must get used to the book keeping of the indices. The concise notation introduced in linear algebra for vector and matrix operations allows expressing difficult problems in a few short equations. This notation can be directly adapted to object oriented programming.

Figure 8-1 shows the classes described in this chapter. Like chapter 2, this chapter discusses some fundamental concepts and operations that shall be used throughout the rest of the book. It might appear austere to many readers because, unlike the preceding chapters, it does not contains concrete examples. However, the reader will find example of use of linear algebra in nearly all remaining chapters of this book.

The chapter begins with a reminder of operations defined on vectors and matrices. Then, two methods for solving systems of linear equations are discussed. This leads to the important concept of matrix inversion. Finally the chapter closes of the problem of finding eigenvalues and eigenvectors.

[1] Space is not to be found; it must always be constructed.

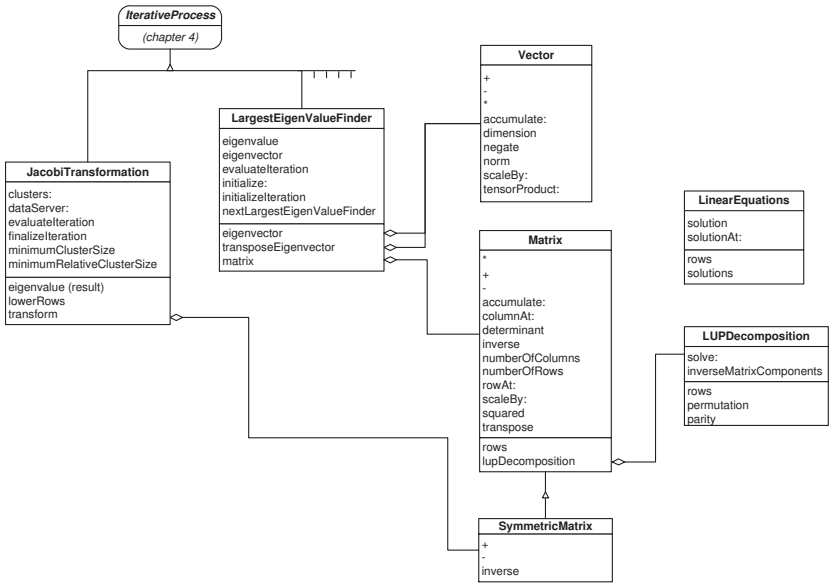


Figure 8-1 Linear algebra classes

8.1 Vectors and matrices

Linear algebra concerns itself with vectors in multidimensional spaces and the properties of operations on these vectors. It is a remarkable fact that such properties can be studied without explicit specification of the space dimension[2].

A vector is an object in a multidimensional space. It is represented by its components measured on a reference system. A reference system is a series of vectors from which the entire space can be generated. A commonly used mathematical notation for a vector is a lower case bold letter, **v** for example. If the set of vectors $\mathbf{u}_1, \ldots, \mathbf{u}_n$ is a reference system for a space with n dimension, then any vector of the space can be written as:

$$\mathbf{v} = v_1\mathbf{u}_1 + \cdots + v_n\mathbf{u}_n, \tag{8.1}$$

where $v_1, \ldots, v_n$ are real numbers in the case of a real space or complex numbers in a complex space. The numbers $v_1, \ldots v_n$ are called the components of the vector.

A matrix is a linear operator over vectors from one space to vectors in another space not necessarily of the same dimension. This means that the application of a matrix on a vector is another vector. To explain what *linear* means, we must quickly introduce some notation.

[2] In fact, most mathematical properties discussed in this chapter are valid for space with an infinite number of dimensions (Hilbert spaces).

A matrix is commonly represented with an upper case bold letter, **M** for example. The application of the matrix **M** on the vector **M** is denoted by $\mathbf{M} \cdot \mathbf{v}$. The fact that a matrix is a linear operator means that

$$\mathbf{M} \cdot (\alpha\mathbf{u} + \beta\mathbf{v}) = \alpha\mathbf{M} \cdot \mathbf{u} + \beta\mathbf{M} \cdot \mathbf{v}, \tag{8.2}$$

for any matrix **M**, any vectors **u** and **v**, any numbers α and β.

Matrices are usually represented using a table of numbers. In general, the number of rows and the number of columns are not the same. A square matrix is a matrix having the same number of rows and columns. A square matrix maps a vector onto another vector of the same space.

Vectors and matrices have an infinite number of representations depending on the choice of reference system. Some properties of matrices are independent from the reference system. Very often the reference system is not specified explicitly. For example, the vector **v** of equation 8.1 is represented by the array of numbers $(v_1 v_2 \cdots v_n)$ where n is the dimension of the vector. Writing the components of a vector within parentheses is customary. Similarly a matrix is represented with a table of numbers called the components of the matrix; the table is also enclosed within parentheses. For example, the n by m matrix **A** is represented by:

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}. \tag{8.3}$$

The components can be real or complex numbers. In this book we shall deal only with vectors and matrices having real components.

For simplicity a matrix can also be written with matrix components. That is, the n by m matrix **A** can be written in the following form:

$$\mathbf{A} = \begin{pmatrix} \mathbf{B} & \mathbf{C} \\ \mathbf{D} & \mathbf{E} \end{pmatrix}, \tag{8.4}$$

where **B**, **C**, **D** and **E** are all matrices of lower dimensions. Let **B** be a p by q matrix. Then, **C** is a p by $m - q$ matrix, **D** is a $n - p$ by q matrix and **E** is a $n - p$ by $m - q$ matrix.

Using this notation one can carry all conventional matrix operations using formulas similar to those written with the ordinary number components. There is, however, one notable exception: the order of the products must be preserved since matrix multiplication is not commutative. For example, the product between two matrices expressed as matrix components can be carried out as:

$$\begin{pmatrix} \mathbf{B}_1 & \mathbf{C}_1 \\ \mathbf{D}_1 & \mathbf{E}_1 \end{pmatrix} \cdot \begin{pmatrix} \mathbf{B}_2 & \mathbf{C}_2 \\ \mathbf{D}_2 & \mathbf{E}_2 x \end{pmatrix} = \begin{pmatrix} \mathbf{B}_1 \cdot \mathbf{B}_2 + \mathbf{C}_1 \cdot \mathbf{D}_2 & \mathbf{B}_1 \cdot \mathbf{C}_2 + \mathbf{C}_1 \cdot \mathbf{E}_2 \\ \mathbf{D}_1 \cdot \mathbf{B}_2 + \mathbf{E}_1 \cdot \mathbf{D}_2 & \mathbf{D}_1 \cdot \mathbf{C}_2 + \mathbf{E}_1 \cdot \mathbf{E}_2 \end{pmatrix}, \tag{8.5}$$

In equation 8.5 the dimension of the respective matrix components must permit the corresponding product. For example the number of rows of the matrices $\mathbf{B}_1$ and $\mathbf{C}_1$ must be equal to the number of columns of the matrices $\mathbf{B}_2$ and $\mathbf{C}_2$ respectively.

Common operations defined on vectors and matrices are summarized below. In each equation, the left-hand side shows the vector notation and the right-hand side shows the expression for coordinates and components.

The sum of two vectors of dimension n is a vector of dimension n:

$$\mathbf{w} = \mathbf{u} + \mathbf{v} \qquad w_i = u_i + v_i \qquad \text{for } i = 1, \ldots, n \tag{8.6}$$

The product of a vector of dimension n by a number α is a vector of dimension n:

$$\mathbf{w} = \alpha\mathbf{v} \qquad w_i = \alpha v_i \qquad \text{for } i = 1, \ldots, n \tag{8.7}$$

The scalar product of two vectors of dimension n is a number:

$$s = \mathbf{u} \cdot \mathbf{v} \qquad s = \sum_{i=1}^{n} u_i v_i \tag{8.8}$$

The norm of a vector is denoted $|\mathbf{v}|$. The norm is the square root of the scalar product with itself.

$$|\mathbf{v}| = \sqrt{\mathbf{v} \cdot \mathbf{v}} \qquad |\mathbf{v}| = \sqrt{\sum_{i=1}^{n} v_i v_i} \tag{8.9}$$

The tensor product of two vectors of respective dimensions n and m is an n by m matrix:

$$\mathbf{T} = \mathbf{u} \otimes \mathbf{v} \qquad T_{ij} = u_i v_j \qquad \begin{array}{l}\text{for } i = 1, \ldots, n \\ \text{and } j = 1, \ldots, m\end{array} \tag{8.10}$$

The sum of two matrices of same dimensions is a matrix of same dimensions:

$$\mathbf{C} = \mathbf{A} + \mathbf{B} \qquad c_{ij} = a_{ij} + b_{ij} \qquad \begin{array}{l}\text{for } i = 1, \ldots, n \\ \text{and } j = 1, \ldots, m\end{array} \tag{8.11}$$

The product of a matrix by a number α is a matrix of same dimensions:

$$\mathbf{B} = \alpha\mathbf{A} \qquad b_{ij} = \alpha a_{ij} \qquad \begin{array}{l}\text{for } i = 1, \ldots, n \\ \text{and } j = 1, \ldots, m\end{array} \tag{8.12}$$

The transpose of a n by m matrix is a m by n matrix:

$$\mathbf{B} = \mathbf{A}^{\mathrm{T}} \qquad b_{ij} = a_{ji} \qquad \begin{array}{l}\text{for } i = 1, \ldots, n \\ \text{and } j = 1, \ldots, m\end{array} \tag{8.13}$$

The product of a n by m matrix with a vector of dimension n is a vector of dimension m:

$$\mathbf{u} = \mathbf{A} \cdot \mathbf{v} \qquad u_i = \sum_{i=1}^{n} a_{ij} v_i \qquad \text{for } i = 1, \ldots, m \tag{8.14}$$

The transposed product of a vector of dimension m by a n by m matrix is a vector of dimension n:

$$\mathbf{u} = \mathbf{v} \cdot \mathbf{A} \qquad u_i = \sum_{i=1}^{m} a_{ji} v_i \qquad \text{for } i = 1, \ldots, m \tag{8.15}$$

The product of a n by p matrix with a p by m matrix is a n by m matrix:

$$\mathbf{C} = \mathbf{A} \cdot \mathbf{B} \qquad c_{ij} = \sum_{k=1}^{p} a_{ik} a_{kj} \qquad \begin{array}{l}\text{for } i = 1, \ldots, m \\ \qquad\quad j = 1, \ldots, m\end{array} \tag{8.16}$$

There are of course other operations (the outer product for example) but they will not be used in this book.

To conclude this quick introduction, let us mention matrices with special properties.

A square matrix is a matrix which has the same number of rows and columns. To shorten sentences, we shall speak of a square matrix of dimension n instead of a n by n matrix.

An identity matrix **I** is a matrix such that

$$\mathbf{I} \cdot \mathbf{v} = \mathbf{v} \tag{8.17}$$

for any vector **v**. This implies that the identity matrix is a square matrix. the representation of the identity matrix contains 1 in the diagonal and 0 off the diagonal in any system of reference. For any square matrix **A** we have:

$$\mathbf{I} \cdot \mathbf{A} = \mathbf{A} \cdot \mathbf{I} = \mathbf{A} \tag{8.18}$$

One important property for the algorithms discussed in this book is symmetry. A symmetrical matrix is a matrix such that $\mathbf{A}^{\mathrm{T}} = \mathbf{A}$. In any system of reference the components of a symmetric matrix have the following property:

$$a_{ij} = a_{ji}, \quad \text{for all } i \text{ and } j. \tag{8.19}$$

The sum and product of two symmetric matrices is a symmetric matrix. The matrix $\mathbf{A}^{\mathrm{T}} \cdot \mathbf{A}$ is a symmetric matrix for any matrix $\mathbf{A}$. If the matrix $\mathbf{A}$ represented in equation 8.4 is symmetric, we have $\mathbf{D} = \mathbf{C}^{\mathrm{T}}$.

Vector and matrix implementation

Listings **??** and 9-9 show respectively the implementation of vectors and matrices. A special implementation for symmetric matrices is shown in listing **??**.

The public interface is designed as to map itself as close as possible to the mathematical definitions. Here are some example of code using operations between vectors and matrices:

```
| u v w a b c |
u := #(1 2 3) asPMVector.
v := #(3 4 5) asPMVector.
a := PMMatrix rows: #((1 0 1) (-1 -2 3)).
b := PMMatrix rows: #((1 2 3) (-2 1 7) (5 6 7)).
w := 4 * u + (3 * v).
c := a * b.
v := a * u.
w := c transpose * v.
w := v * c
```

In the first two lines after the declarative statement, the vectors **u** and **v** are defined from their component array using the creator method `asPMVector`. They are 3-dimensional vectors. The matrices **a** and **b** are created by supplying the components to the class creation method `rows:`. The matrix **a** is a 2 by 3 matrix, whereas the matrix **b** is a square matrix of dimension 3. In all cases the variable `w` is assigned to a vector and the variable `c` is assigned to a matrix. First, the vector **w** is assigned to a linear combination of the vectors **u** and **v**. Apart from the parentheses required for the second product, the expression is identical to what one would write in mathematics (compare this expression with equation 8.1).

Next the matrix **c** is defined as the product of the matrices **a** and **b** in this order. It is a direct transcription of the left part of equation 8.16 up to the case of the operands.

The next assignment redefines the vector **v** as the product of the matrix **A** with the vector **u**. It is now a 2-dimensional vector. Here again the correspondence between the Pharo and the mathematical expression is direct.

The last two lines compute the vector **w** as the transpose product with the matrix **a**. The result of both line is the same[3]. The first line makes the trans-

[3]There is a subtle difference between regular vectors and transposed vectors, which is overlooked by our choice of implementation, however. Transposed vectors or covariant vectors as they are called in differential geometry should be implemented in a proper class. This extension is left as an exercise to the reader.

position of the matrix **a** explicit, whereas the second line used the implicit definition of the transpose product. The second line is faster than the previous one since no memory assignment is required for the temporary storage of the transpose matrix.

The use of other methods corresponding to the operations defined in equations 8.6 to 8.16 are left as an exercise to the reader.

Implementation

A vector is akin to an instance of the Pharo class `Array`, for which mathematical operations have been defined. Thus, a vector in Pharo can be implemented directly as a subclass of the class `Array`. A matrix is an object whose instance variable is an array of vectors.

The operations described in the preceding section can be assigned to the corresponding natural operators. The multiplication, however, can involve several types of operands. It can be applied between

1. a vector and a number,
2. a matrix and a number or
3. a vector and a matrix.

Thus, the multiplication will be implemented using double dispatching as explained in section 2.3 for operations between polynomials. Double dispatching is described in appendix (*c.f.* section **??**).

The method `asPMVector` is defined for compatibility with a similar method defined in the class `Collection` to construct a vector out of any collection object.

The method `tensorProduct` returns an instance of a symmetric matrix. This class is defined in listing **??**.

The method `accumulate` is meant to be used when there is a need to add several vectors. Indeed the following code

```
| a b c d e |
a := #(1 2 3 4 5) asPMVector.
b := #(2 3 4 5 6) asPMVector.
c := #(3 4 5 6 7) asPMVector.
d := #(4 5 6 7 8) asPMVector.
e := a+b+c+d.
```

creates a lots of short-lived vectors, namely one for each addition. Using the method `accumulate` reduces the memory allocation:

```
| a b c d e |
a := #(1 2 3 4 5) asPMVector.
b := #(2 3 4 5 6) asPMVector.
c := #(3 4 5 6 7) asPMVector.
```

```
d := #(4 5 6 7 8) asPMVector.
e := a copy.
e accumulate: b; accumulate: c; accumulate: d.
```

If vectors of large dimension are used, using accumulation instead of addition can make a big difference in performance since many large short-lived objects put a heavy toll of the garbage collector.

Listing 8-2 Vector class in Pharo

```
Array variableSubclass: #PMVector
   instanceVariableNames: ''
   classVariableNames: ''
   package: 'Math-Core'
```

```
PMVector >> * aNumberOrMatrixOrVector
    ^aNumberOrMatrixOrVector productWithVector: self
```

```
PMVector >> + aVector
    | answer n |
    answer := self class new: self size.
    n := 0.
    self with: aVector do:
        [ :a :b |
          n := n + 1.
          answer at: n put: ( a + b).
        ].
    ^answer
```

```
PMVector >> - aVector
    | answer n |
    answer := self class new: self size.
    n := 0.
    self with: aVector do:
        [ :a :b |
          n := n + 1.
          answer at: n put: ( a - b).
        ].
    ^answer
```

```
PMVector >> accumulate: aVectorOrAnArray
    1 to: self size do: [ :n | self at: n put: ( ( self at: n) + (
                                          aVectorOrAnArray at:
    n))].
```

```
PMVector >> accumulateNegated: aVectorOrAnArray
    1 to: self size do: [ :n | self at: n put: ( ( self at: n) - (
                                          aVectorOrAnArray at:
    n))].
```

```
PMVector >> asVector
    ^ self
```

```
PMVector >> dimension
    ^ self size
```

```
PMVector >> negate
    1 to: self size do: [ :n | self at: n put: (self at: n) negated].
```

```
PMVector >> norm
    ^ (self * self) sqrt
```

```
PMVector >> normalized
    ^ (1 / self norm) * self
```

```
PMVector >> productWithMatrix: aMatrix
    ^ aMatrix rowsCollect: [ :each | each * self ]
```

```
PMVector >> productWithVector: aVector
    | n |
    n := 0.
    ^self inject: 0
            into: [ :sum :each | n := n + 1. (aVector at: n) * each +

    sum ]
```

```
PMVector >> scaleBy: aNumber
    1 to: self size do: [ :n | self at: n put: ( ( self at: n) *
                                                        aNumber)
    ].
```

```
PMVector >> tensorProduct: aVector
    self dimension = aVector dimension
        ifFalse:[ ^self error: 'Vector dimensions mismatch to build
                                                    tensor
    product'].
    ^PMSymmetricMatrix rows: ( self collect: [ :a | aVector collect:
                                                    [ :b | a *
    b]])
```

The class `PMMatrix` has two instance variables:

- `rows` an array of vectors, each representing a row of the matrix and
- `lupDecomposition` a pointer to an object of the class `PMLUPDecomposition` containing the LUP decomposition of the matrix if already computed. LUP decomposition is discussed in section 8.3.

This implementation reuses the vector implementation of the vector scalar product to make the code as compact as possible. The iterator methods `columnsCollect:`, `columnsDo:`, `rowsCollect:` and `rowsDo:` are designed to limit the need for index management to these methods only.

An attentive reader will have noticed that the iterator methods `rowsDo:` and `rowsCollect:` present a potential breach of encapsulation. Indeed, the following expression

```
aMatrix rowsDo:[ :each | each at: 1 put: 0]
```

changes the matrix representation outside of the normal way. Similarly, the expression

```
aMatrix rowsCollect:[ :each | each]
```

gives direct access to the matrix's internal representation.

The method `square` implements the product of the transpose of a matrix with itself. This construct is used in several algorithms presented in this book.

> **Note:** The presented matrix implementation is straightforward. Depending on the problem to solve, however, it is not the most efficient one. Each multiplication allocates a lot of memory. If the problem is such that one can allocate memory once for all, more efficient methods can be designed.

The implementation of matrix operations — addition, subtraction, product — uses double or multiple dispatching to determine whether or not the result is a symmetric matrix. Double and multiple dispatching are explained in sections **??** and **??**. The reader who is not familiar with multiple dispatching should trace down a few examples between simple matrices using the debugger.

Listing 8-3 Matrix classes

```
Object subclass: #PMMatrix
   instanceVariableNames: 'rows lupDecomposition'
   classVariableNames: ''
   package: 'Math-Matrix'
```

```
PMMatrix class >> new: anInteger
    ^ self new initialize: anInteger
```

```
PMMatrix class >> rows: anArrayOrVector
    ^ self new initializeRows: anArrayOrVector
```

```
PMMatrix >> * aNumberOrMatrixOrVector
    ^ aNumberOrMatrixOrVector productWithMatrix: self
```

```
PMMatrix >> + aMatrix
    ^ aMatrix addWithRegularMatrix: self
```

```
PMMatrix >> - aMatrix
    ^ aMatrix subtractWithRegularMatrix: self
```

```
PMMatrix >> accumulate: aMatrix
    | n |
    n := 0.
    self rowsCollect: [ :each | n := n + 1. each accumulate: (
                                                    aMatrix rowAt:
    n)]
```

```
PMMatrix >> accumulateNegated: aMatrix
    | n |
    n := 0.
    self rowsCollect: [ :each | n := n + 1. each accumulateNegated: (
                                                        aMatrix rowAt:
    n)]
```

```
PMMatrix >> addWithMatrix: aMatrix class: aMatrixClass
    | n |
    n := 0.
    ^ aMatrixClass rows: ( self rowsCollect: [ :each | n := n + 1.
                                            each + ( aMatrix rowAt:
    n)])
```

```
PMMatrix >> addWithRegularMatrix: aMatrix
    ^ aMatrix addWithMatrix: self class: aMatrix class
```

```
PMMatrix >> addWithSymmetricMatrix: aMatrix
    ^ aMatrix addWithMatrix: self class: self class
```

```
PMMatrix >> asSymmetricMatrix
    ^PMbSymmetricMatrix rows: rows
```

```
PMMatrix >> columnAt: anInteger
    ^ rows collect: [ :each | each at: anInteger]
```

```
PMMatrix >> columnsCollect: aBlock
    | n |
    n := 0.
    ^rows last collect: [ :each | n := n + 1. aBlock value: ( self
                                                        columnAt:
    n)]
```

```
PMMatrix >> columnsDo: aBlock
    | n |
    n := 0.
    ^ rows last do: [ :each | n := n + 1. aBlock value: ( self
                                                        columnAt:
    n)]
```

```
PMMatrix >> initialize: anInteger
    rows := ( 1 to: anInteger) asVector collect: [ :each | PMVector
                                                        new:
    anInteger].
```

```
PMMatrix >> initializeRows: anArrayOrVector
    rows := anArrayOrVector asVector collect: [ :each | each

    asVector].
```

```
PMMatrix >> isSquare
    ^ rows size = rows last size
\end{verbatim}
```

```
\begin{displaycode}{Smalltalk}
PMMatrix >> isSymmetric
    ^ false
```

```
PMMatrix >> lupDecomposition
    lupDecomposition isNil
        ifTrue: [ lupDecomposition := PMLUPDecomposition equations:

    rows].
    ^ lupDecomposition
```

```
PMMatrix >> negate
    rows do: [ :each |each negate].
```

```
PMMatrix >> numberOfColumns
    ^ rows last size
```

```
PMMatrix >> numberOfRows
    ^ rows size
```

```
PMMatrix >> printOn: aStream
    | first |
    first := true.
    rows do:
        [ :each |
          first ifTrue: [ first := false]
                ifFalse:[ aStream cr].
          each printOn: aStream.
        ].
```

```
PMMatrix >> productWithMatrix: aMatrix
    ^ self productWithMatrixFinal: aMatrix
```

```
PMMatrix >> productWithMatrixFinal: aMatrix
    ^ self class rows: ( aMatrix rowsCollect: [ :row | self
                                columnsCollect: [ :col | row *
    col]])
```

```
PMMatrix >> productWithSymmetricMatrix: aSymmetricMatrix
    ^ self class rows: ( self rowsCollect: [ :row | aSymmetricMatrix
                                columnsCollect: [ :col | row *
    col]])
```

```
PMMatrix >> productWithTransposeMatrix: aMatrix
    ^ self class rows: ( self rowsCollect: [ :row | aMatrix
                                    rowsCollect: [ :col | row *
    col]])
```

```
PMMatrix >> productWithVector: aVector
    ^ self columnsCollect: [ :each | each * aVector]
```

```
PMMatrix >> rowAt: anInteger
    ^ rows at: anInteger
```

```
PMMatrix >> rowsCollect: aBlock
    ^ rows collect: aBlock
```

```
PMMatrix >> rowsDo: aBlock
    ^ rows do: aBlock
```

```
PMMatrix >> scaleBy: aNumber
    rows do: [ :each | each scaleBy: aNumber].
```

```
PMMatrix >> squared
    ^ PMSymmetricMatrix rows: ( self columnsCollect: [ :col | self
                                columnsCollect: [ :colT | col *
    colT]])
```

```
PMMatrix >> subtractWithMatrix: aMatrix class: aMatrixClass
    | n |
    n := 0.
    ^ aMatrixClass rows: ( self rowsCollect: [ :each | n := n + 1.
                                        each - ( aMatrix rowAt:
    n)])
```

```
PMMatrix >> subtractWithRegularMatrix: aMatrix
    ^ aMatrix subtractWithMatrix: self class: aMatrix class
```

```
PMMatrix >> subtractWithSymmetricMatrix: aMatrix
    ^ aMatrix subtractWithMatrix: self class: self class
```

```
PMMatrix >> transpose
    ^ self class rows: ( self columnsCollect: [ :each | each])
```

```
PMMatrix >> transposeProductWithMatrix: aMatrix
    ^ self class rows: ( self columnsCollect: [ :row | aMatrix
                                columnsCollect: [ :col | row *
    col]])
```

Listing **??** shows the implementation of the class `PMSymmetricMatrix` representing symmetric matrices. A few algorithms are implemented differently for symmetric matrices.

The reader should pay attention to the methods implementing addition, subtraction and products. Triple dispatching is used to ensure that the addition or subtraction of two symmetric matrices yields a symmetric matrix whereas the same operations between a symmetric matrix and a normal matrix yield a normal matrix. Product requires quadruple dispatching.

Listing 8-4 Symmetric matrix classes

```
PMMatrix subclass: #SymmetricMatrix
   instanceVariableNames: ''
   classVariableNames: ''
   package: 'Math-Matrix'
```

```
SymmetricMatrix class >> identity: anInteger
    ^ self new initializeIdentity: anInteger
```

```
SymmetricMatrix >> + aMatrix
    ^ aMatrix addWithSymmetricMatrix: self
```

```
SymmetricMatrix >> - aMatrix
    ^ aMatrix subtractWithSymmetricMatrix: self
```

```
SymmetricMatrix >> addWithSymmetricMatrix: aMatrix
    ^ aMatrix addWithMatrix: self class: self class
```

```
SymmetricMatrix >> clear
    rows do: [ :each | each atAllPut: 0].
```

```
SymmetricMatrix >> initializeIdentity: anInteger
    rows := ( 1 to: anInteger) asVector collect: [ :n | (DhbVector
                new: anInteger) atAllPut: 0; at: n put: 1;
    yourself].
```

```
SymmetricMatrix >> isSquare
    ^ true
```

```
SymmetricMatrix >> isSymmetric
    ^ true
```

```
SymmetricMatrix >> productWithMatrix: aMatrix
    ^ aMatrix productWithSymmetricMatrix: self
```

```
SymmetricMatrix >> productWithSymmetricMatrix: aSymmetricMatrix
    ^ aSymmetricMatrix productWithMatrixFinal: self
```

```
SymmetricMatrix >> subtractWithSymmetricMatrix: aMatrix
    ^ aMatrix subtractWithMatrix: self class: self class
```

8.2 Linear equations

A linear equation is an equation in which the unknowns appear to the first order and are combined with the other unknowns only with addition or subtraction. For example, the following equation:

$$3x_1 - 2x_2 + 4x_3 = 0, \tag{8.20}$$

is a linear equation for the unknowns x_1, x_2 and x_3. The following equation

$$3x_1^2 - 2x_2 + 4x_3 - 2x_2x_3 = 0, \tag{8.21}$$

is not linear because x_1 appears as a second order term and a term containing the product of the unknowns x_2 and x_3 is present. However, equation 8.21 is linear for the unknown x_2 (or x_3) alone. A system of linear equation has the same number of equations as there are unknowns. For example

$$\begin{cases} 3x_1 + 2y_2 + 4z_3 &= 16 \\ 2x_1 - 5y_2 - z_3 &= 6 \\ x_1 - 2y_2 - 2z_3 &= 10 \end{cases} \tag{8.22}$$

is a system of linear equation which can be solved for the 3 unknowns x_1, x_2 and x_3. Its solution is $x_1 = 2$, $x_2 = -1$ and $x_3 = 3$.

A system of linear equations can be written in vector notation as follows:

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{y}. \tag{8.23}$$

The matrix **A** and the vector **z** are given. Solving the system of equations is looking for a vector **x** such that equation 8.23 holds. The vector **x** is the solution of the system. A necessary condition for a unique solution to exist is that the matrix **A** be a square matrix. Thus, we shall only concentrate on square matrices[4]. A sufficient condition for the existence of a unique solution is that the rank of the matrix — that is the number of linearly independent rows — is equal to the dimension of the matrix. If the conditions are all fulfilled, the solution to equation 8.23 can be written in vector notation:

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{y}. \tag{8.24}$$

where $\mathbf{A}^{-1}$ denotes the inverse of the matrix **A** (*c.f.* section 8.5).

Computing the inverse of a matrix in the general case is numerically quite demanding (*c.f.* section 8.5). Fortunately, there is no need to explicitly compute the inverse of a matrix to solve a system of equations. Let us first assume that the matrix of the system is a triangular matrix, that is we have:

$$\mathbf{T}\mathbf{x} = \mathbf{y}'. \tag{8.25}$$

where **T** is a matrix such that:

$$T_{ij} = 0 \quad \text{for } i > j. \tag{8.26}$$

Backward substitution

The solution of the system of equation 8.25 can be obtained using backward substitution. The name backward comes from the fact that the solution begins by calculating the component with the highest index; then it works its way backward on the index calculating each components using all components with higher index.

$$\begin{cases} x_n = \dfrac{y'_n}{t_{nn}}, \\ x_i = \dfrac{y'_i - \displaystyle\sum_{j=i+1}^{n} t_{ij}x_j}{a'_{nn}} \quad \text{for } i = n-1, \ldots, 1. \end{cases} \tag{8.27}$$

[4] It is possible to solve system of linear equations defined with a non-square matrix using technique known as singular value decomposition (SVD). In this case, however, the solution of the system is not a unique vector, but a subspace of n-dimensional space where n is the number of columns of the system's matrix. The SVD technique is similar to the techniques used to find eigenvalues and eigenvectors.

Gaussian elimination

Any system as described by equation 8.23 can be transformed into a system based on a triangular matrix. This can be achieved through a series of transformations leaving the solution of the system of linear equations invariant. Let us first rewrite the system under the form of a single matrix **S** defined as follows:

$$\mathbf{S} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} & y_1 \\ a_{21} & a_{22} & \dots & a_{2n} & y_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} & y_n \end{pmatrix}. \tag{8.28}$$

Among all transformations leaving the solution of the system represented by the matrix **S** invariant, there are two transformations, which help to obtain a system corresponding to triangular matrix. First, any row of the matrix **S** can be exchanged. Second, a row of the matrix **S** can be replaced by a linear combination[5] of that row with another one. The trick is to replace all rows of the matrix **S**, except for the first one, with rows having a vanishing first coefficient.

If $a_{11} = 0$, we permute the first row with row i such that $a_{i1} \neq 0$. Then, we replace each row j, where $j > 1$, by itself subtracted with the first row multiplied by a_{i1}/a_{11} This yields a new system matrix $\mathbf{S}'$ of the form:

$$\mathbf{S}' = \begin{pmatrix} a'_{11} & a'_{12} & \dots & a'_{1n} & y'_1 \\ 0 & a'_{22} & \dots & a'_{2n} & y'_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & a'_{n2} & \dots & a'_{nn} & y'_n \end{pmatrix}. \tag{8.29}$$

This step is called *p*ivoting the system on row 1 and the element a_{11} after the permutation is called the pivot. By pivoting the system on the subsequent rows, we shall obtain a system built on a triangular matrix as follows:

$$\mathbf{S}^{(n)} = \begin{pmatrix} a_{11}^{(n)} & a_{12}^{(n)} & a_{13}^{(n)} & \dots & a_{1n}^{(n)} & y_1^{(n)} \\ 0 & a_{22}^{(n)} & a_{23}^{(n)} & \dots & a_{2n}^{(n)} & y_2^{(n)} \\ 0 & 0 & a_{33}^{(n)} & \dots & a_{3n}^{(n)} & y_3^{(n)} \\ \vdots & \vdots & & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 0 & a_{nn}^{(n)} & y_n^{(n)} \end{pmatrix}. \tag{8.30}$$

This algorithm is called Gaussian elimination. Gaussian elimination will fail if we are unable to find a row with a non-zero pivot at one of the steps. In that case the system does not have a unique solution .

The first n columns of the matrix $\mathbf{S}^{(n)}$ can be identified to the matrix **T** of equation 8.25 and the last column of the matrix $\mathbf{S}^{(n)}$ corresponds to the vector $\mathbf{y}'$ of equation 8.25. Thus, the final system can be solved with backward substitution.

[5] In such linear combination, the coefficient of the replaced row must not be zero.

> **Note:** The reader can easily understand that one could have made a transformation to obtain a triangular matrix with the elements above the diagonal all zero. In this case, the final step is called forward substitution since the first component of the solution vector is computed first. The two approaches are fully equivalent.

Fine points

A efficient way to avoid a null pivot is to systematically look for the row having the largest pivot at each step. To be precise, before pivoting row i, it is first exchanged with row j such that $\left|a_{ij}^{(i-1)}\right| > \left|a_{ik}^{(i-1)}\right|$ for all $k = i, \ldots, n$ if such row exists. The systematic search for the largest pivot ensures optimal numerical precision [**?**].

The reader will notice that all operations can be performed in place since the original matrix **S** is not needed to compute the final result.

Finally it should be noted that the pivoting step can be performed on several vectors **y** at the same time. If one must solve the same system of equations for several sets of constants, pivoting can be applied to all constant vectors by extending the matrix **S** with as many columns as there are additional constant vectors as follows:

$$\mathbf{S} = \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} & y_{11} & \ldots & y_{m1} \\ a_{21} & a_{22} & \ldots & a_{2n} & y_{12} & \ldots & y_{m2} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \ldots & a_{nn} & y_{1n} & \ldots & y_{mn} \end{pmatrix}. \tag{8.31}$$

backward substitution must of course be evaluated separately for each constant vector.

Gaussian elimination is solely dedicated to solving systems of linear equations. The algorithm is somewhat slower than LUP decomposition described in section 8.3. When applied to systems with several constant vectors, however, Gaussian elimination is faster since the elimination steps are made only once. In the case of LUP decomposition, obtaining the solution for each vector requires more operations than those needed by backward substitution.

Linear equations — Smalltalk implementation

Although using matrix and vector notation greatly simplifies the discussion of Gaussian elimination, there is little gain in making an implementation using matrices and vectors explicitly.

The class creation methods or constructors will take as arguments either a matrix and a vector or an array of arrays and an array. The class implementing Gaussian elimination has the following instance variables:

- `rows` an array or a vector whose elements contain the rows of the matrix **S**.
- `solutions` an array whose elements contain the solutions of the system corresponding to each constant vector.

Solving the system is entirely triggered by retrieving the solutions. The instance variable `solutions` is used to keep whether or not Gaussian elimination has already taken place. If this variable is `nil` Gaussian elimination has not yet been performed. Gaussian elimination is performed by the method `solve`. At the end of the algorithm, the vector of solutions is allocated into the instance variable `solutions`. Similarly, backward substitution is triggered by the retrieving of a solution vector. If the solution for the specified index has not yet been computed, backward substitution is performed and the result is stored in the solution array.

Listing 8-5 shows the class `PMLinearEquationSystem` implementing Gaussian elimination.

To solve the system of equations 8.22 using Gaussian elimination, one needs to write the following expression:

```
(PMLinearEquationSystem equations: #((3 2 4)
                                     (2 -5 -1)
                                     (1 -2 2))
                       constant: #(16 6 10)
    ) solution.
```

This expression has been written on three lines to delineate the various steps. The first two lines create an instance of the class `PMLinearEquationSystem` by feeding the coefficients of the equations, rows by rows, on the first line and giving the constant vector on the second line. The last line is a call to the method `solution` retrieving the solution of the system.

Solving the same system with an additional constant vector requires a little more code, but not much:

```
| s sol1 sol2 |
s := PMLinearEquationSystem equations: #((3 2 4) (2 -5 -1) (1 -2 2))
                         constants: #((16 6 10)
                                      (7 10 9)).
sol1 := s solutionAt: 1.
sol2 := s solutionAt: 2.
```

In this case, the creation method differs in that the two constant vectors are supplied in an array columns by columns. Similarly, the two solutions must be fetched one after the other.

The class `PMLinearEquationSystem`. The class method `equations:constants:` allows one to create a new instance for a given matrix and a series of constant vectors.

The method `solutionAt:` returns the solution for a given constant vector. The index specified as argument to that method corresponds to that of the desired constant vector.

The method `solve` performs all required pivoting steps using a `do:` iterator. The method `pivotStepAt:` first swaps rows to bring a pivot having the largest absolute value in place and then invokes the method `pivotAt:` to perform the actual pivoting.

Convenience methods `equations:constant:` and `solution` are supplied to treat the most frequent case where there is only one single constant vector. However, the reader should be reminded that LUP decomposition is more effective in this case.

If the system does not have a solution — that is, if the system's matrix is singular — an arithmetic error occurs in the method `pivotAt:` when the division with the zero pivot is performed. The method `solutionAt:` traps this error within an exception handling structure and sets the solution vector to a special value — the integer 0 — as a flag to prevent attempting Gaussian elimination a second time. Then, the value `nil` is returned to represent the non-existent solution.

Listing 8-5 Implementation of a system of linear equations

```
Object variableSubclass: #PMLinearEquationSystem
   instanceVariableNames: 'rows solutions'
   classVariableNames: ''
   package: 'Math-Core'
```

```
PMLinearEquationSystem class >> equations: anArrayOfArrays constant:
    anArray
  ^ self basicNew
      initialize: anArrayOfArrays
      constants: (Array with: anArray)
```

```
PMLinearEquationSystem class >> equations: anArrayOfArrays
    constants: anArrayOfConstantArrays
  ^ self basicNew
      initialize: anArrayOfArrays
      constants: anArrayOfConstantArrays
```

```
PMLinearEquationSystem >> backSubstitutionAt: anInteger
    | size answer accumulator |
    size := rows size.
    answer := Array new: size.
    size to: 1 by: -1 do:
        [ :n |
          accumulator := (rows at: n) at: (anInteger + size).
          ( n + 1) to: size
            do: [ :m | accumulator := accumulator - ( ( answer at: m)
                                        * ( ( rows at: n) at:
    m))].
```

```
            answer at: n put: ( accumulator / ( ( rows at: n) at: n))
        ].
        solutions at: anInteger put: answer
```

```
PMLinearEquationSystem >> initialize: anArrayOfArrays constants:
        anArrayOfConstantArrays
        | n |
        n := 0.
        rows := anArrayOfArrays collect: [ :each | n := n + 1. each, (
                        anArrayOfConstantArrays collect: [ :c | c at: n])]
```

```
PMLinearEquationSystem >> largestPivotFrom: anInteger
        | valueOfMaximum indexOfMaximum |
        valueOfMaximum := ( rows at: anInteger) at: anInteger.
        indexOfMaximum := anInteger.
        ( anInteger + 2) to: rows size do:
            [ :n |
              ( ( rows at: n) at: anInteger) > valueOfMaximum
                    ifTrue: [ valueOfMaximum := ( rows at: n) at:

        anInteger.
                                  indexOfMaximum := n ].
            ].
        ^indexOfMaximum
```

```
PMLinearEquationSystem >> pivotAt: anInteger
        | inversePivot rowPivotValue row pivotRow |
        pivotRow := rows at: anInteger.
        inversePivot := 1 / ( pivotRow at: anInteger).
        ( anInteger + 1) to: rows size do:
            [ :n |
              row := rows at: n.
              rowPivotValue := ( row at: anInteger) * inversePivot.
              anInteger to: row size do:
                [ :m |
                  row at: m put: ( ( row at: m) - (( pivotRow at: m) *

        rowPivotValue)) ].
            ]
```

```
PMLinearEquationSystem >> pivotStepAt: anInteger
        self swapRow: anInteger withRow: ( self largestPivotFrom:

        anInteger);
               pivotAt: anInteger
```

```
PMLinearEquationSystem >> printOn: aStream
        | first delimitingString n k |
        n := rows size.
        first := true.
        rows do:
            [ :row |
```

```
          first ifTrue: [ first := false ]
                 ifFalse: [ aStream cr ].
          delimitingString := '('.
          k := 0.
          row do:
            [ :each |
                aStream nextPutAll: delimitingString.
                each printOn: aStream.
                k := k + 1.
                delimitingString := k < n ifTrue: [ ' '] ifFalse: [ '
                                                                    :
    '].
            ].
          aStream nextPut: $).
        ]
```

```
PMLinearEquationSystem >> solution
    ^self solutionAt: 1
```

```
PMLinearEquationSystem >> solutionAt: anInteger
    solutions isNil
        ifTrue: [ [self solve] when: ExError do: [ :signal |solutions
                                        := 0. signal exitWith: nil.]
    ].
    solutions = 0
        ifTrue: [ ^nil].
    ( solutions at: anInteger) isNil
        ifTrue: [ self backSubstitutionAt: anInteger].
    ^ solutions at: anInteger
```

```
PMLinearEquationSystem >> solve
    1 to: rows size do: [ :n | self pivotStepAt: n].
    solutions := Array new: ( (rows at: 1) size - rows size).
```

```
PMLinearEquationSystem >> swapRow: anInteger1 withRow: anInteger2
    | swappedRow |
    anInteger1 = anInteger2
        ifFalse: [ swappedRow := rows at: anInteger1.
                   rows at: anInteger1 put: ( rows at:

    anInteger2).
                   rows at: anInteger2 put: swappedRow ]
```

8.3 LUP decomposition

LUP decomposition is another technique to solve a system of linear equations. It is an alternative to the Gaussian elimination [**?**]. Gaussian elimination can solve a system with several constant vectors, but all constant vectors must be known before starting the algorithm.

LUP decomposition is done once for the matrix of a given system. Thus, the system can be solved for any constant vector obtained after the LUP decomposition. In addition, LUP decomposition gives a way to calculate the determinant of a matrix and it can be used to compute the inverse of a matrix.

LUP stands for Lower, Upper and Permutation. It comes from the observation that any non-singular square matrix $\mathbf{A}$ can be decomposed into a product of 3 square matrices of the same dimension as follows:

$$\mathbf{A} = \mathbf{L} \cdot \mathbf{U} \cdot \mathbf{P}, \tag{8.32}$$

where $\mathbf{L}$ is a matrix whose components located above the diagonal are zero (lower triangular matrix), $\mathbf{U}$ is a matrix whose components located below the diagonal are zero (upper triangular matrix) and $\mathbf{P}$ is a permutation matrix. The decomposition of equation 8.32 is non-unique. One can select a unique decomposition by requiring that all diagonal elements of the matrix $\mathbf{L}$ be equal to 1.

The proof that such decomposition exists is the algorithm itself. It is a proof by recursion. We shall first start to construct an LU decomposition, that is an LUP decomposition with an identity permutation matrix. Let us write the matrix as follows:

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix} = \begin{pmatrix} a_{11} & \mathbf{w}^{\mathrm{T}} \\ \mathbf{v} & \mathbf{A}' \end{pmatrix}, \tag{8.33}$$

where $\mathbf{v}$ and $\mathbf{w}$ are two vectors of dimension $n-1$ and $\mathbf{A}'$ is a square matrix of dimension $n-1$. Written in this form, one can write an LU decomposition of the matrix $\mathbf{A}$ as follows:

$$\mathbf{A} = \begin{pmatrix} a_{11} & \mathbf{w}^{\mathrm{T}} \\ \mathbf{v} & \mathbf{A}' \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ \frac{\mathbf{v}}{a_{11}} & \mathbf{I}_{n-1} \end{pmatrix} \cdot \begin{pmatrix} a_{11} & \mathbf{w}^{\mathrm{T}} \\ 0 & \mathbf{A}' - \frac{\mathbf{v} \otimes \mathbf{w}}{a_{11}} \end{pmatrix}, \tag{8.34}$$

where $\mathbf{I}_{n-1}$ is an identity matrix of dimension $n-1$. The validity of equation 8.34 can be verified by carrying the product of the two matrices of the right-hand side using matrix components as discussed in section 8.1. We now are left with the problem of finding an LU decomposition for the matrix $\mathbf{A}' - \frac{\mathbf{v} \otimes \mathbf{w}}{a_{11}}$. This matrix is called the Shur's complement of the matrix with respect to the pivot element a_{11}. Let us assume that we have found such a decomposition for that matrix, that is, that we have:

$$\mathbf{A}' - \frac{\mathbf{v} \otimes \mathbf{w}}{a_{11}} = \mathbf{L}' \cdot \mathbf{U}'. \tag{8.35}$$

The we have:

$$\begin{pmatrix} 1 & 0 \\ \frac{\mathbf{v}}{a_{11}} & \mathbf{I}_{n-1} \end{pmatrix} \cdot \begin{pmatrix} a_{11} & \mathbf{w}^{\mathrm{T}} \\ 0 & \mathbf{L}' \cdot \mathbf{U}' \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ \frac{\mathbf{v}}{a_{11}} & \mathbf{I}_{n-1} \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 \\ 0 & \mathbf{L}' \end{pmatrix} \cdot \begin{pmatrix} a_{11} & \mathbf{w}^{\mathrm{T}} \\ 0 & \mathbf{U}' \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 0 \\ \frac{\mathbf{v}}{a_{11}} & \mathbf{L}' \end{pmatrix} \cdot \begin{pmatrix} a_{11} & \mathbf{w}^{\mathrm{T}} \\ 0 & \mathbf{U}' \end{pmatrix}. \tag{8.36}$$

The above equality can be verified by carrying the multiplication with matrix components. The second line of equation 8.36 is the LU decomposition of the matrix **A**, which we were looking for. The algorithm is constructed recursively on the successive Shur's complements. For practical implementation, however, it is best to use a loop.

Building the Shur's complement involves a division by the pivot element. As for Gaussian elimination, the algorithm runs into trouble if this element is zero. The expression for the Shur's complement (equations 8.35) shows that, if the pivot element is small, rounding errors occur when computing the elements of the Shur's complement. The strategy avoiding this problem is the same as for Gaussian elimination. One must find the row having the component with the largest absolute value and use this component as the pivot. This is where the permutation matrix comes into play. It will keep track of each row permutation required to bring the row selected for pivoting into the first position. If a non-zero pivot element cannot be found at one step, then the matrix **A** is singular and no solution to the system of equation can be found (*c.f.* similar discussion in section 8.2).

Now that we have proved that an LUP decomposition exists for any non-singular matrix, let us see how it is used to solve a system of linear equations. Consider the system described by equation 8.23; using the LUP decomposition of the matrix **A**, it can be rewritten as:

$$\mathbf{LU} \cdot \mathbf{x} = \mathbf{P} \cdot \mathbf{y}, \tag{8.37}$$

where we have used the fact that $\mathbf{P}^{-1} = \mathbf{P}$ for any permutation matrix. Equation 8.37 can be solved in two steps. First one solves the system

$$\mathbf{L} \cdot \tilde{\mathbf{x}} = \mathbf{P} \cdot \mathbf{y} \tag{8.38}$$

using forward substitution. Then, one solves the system

$$\mathbf{U} \cdot \mathbf{x} = \tilde{\mathbf{x}} \tag{8.39}$$

using backward substitution. One can see that the LUP decomposition can be used several times to solve a linear system of equations with the same matrix and different constant vectors.

Performing LUP decomposition is faster than performing Gaussian elimination because Gaussian elimination must also transform the constant vector.

To compute the solution vector, however, Gaussian elimination only needs backward substitution whereas LUP requires both forward and backward substitution. The end result is that solving a system of linear equation for a single constant vector is slightly faster using LUP decomposition. If one needs to solve the same system of equations for several constant vectors known in advance, Gaussian elimination is faster. If the constant vectors are not known in advance — or cannot be all stored with the original matrix — LUP decomposition is the algorithm of choice.

LUP decomposition — General implementation

To implement the LUP algorithm, let us first note that we do not need much storage for the three matrices **L**, **U** and **P**. Indeed, the permutation matrix **P** can be represented with a vector of integers of the same dimension as the matrix rows. Since the diagonal elements of the matrix **L** are set in advance, we only need to store elements located below the diagonal. These elements can be stored in the lower part of the matrix **U**. Looking at the definition of the Shur's complement (equation 8.35) and at equation 8.36 we can see that all operations can be performed within a matrix of the same size as the original matrix[6].

The implementation of the LUP algorithm must create storage to place the components of the matrix whose LUP decomposition is to be computed. A method implements the solving of the system of equations for a given constant vector. Within the method the LUP decomposition itself is performed if it has not yet been made using lazy initialization. During the computation of the LUP decomposition the parity of the permutation is tracked. This information is used to compute the determinant of the matrix (*c.f.* section 8.4). Thus, the class implementing LUP decomposition has the following instance variables.

- `rows` contains a copy of the rows of the matrix representing the system of linear equations, *i.e.*the matrix **A**; copying the matrix is necessary since LUP decomposition destroys the components; at the end of the LUP decomposition, it will contain the components of the matrices **L** and **U**,
- `permutation` contains an array of integers describing the permutation, *i.e.*the matrix **P**,
- `parity` contains parity of the permutation for efficiency purpose [7].

[6] If the matrix **A** is no longer needed after solving the system of equation, the LUP decomposition can even be performed inside the matrix **A** itself.

[7] The parity is needed to compute the determinant. It could be computed from the parity matrix. However, the overhead of keeping track of the parity is negligible compared to the LUP steps and it is much faster than computing the parity.

The instance variable `permutation` is set to undefined (`nil` in Smalltalk) at initialization time by default. It is used to check whether the decomposition has already been made or not.

The method `solve` implements the solving of the equation system for a given constant vector. It first checks whether the LUP decomposition has been performed. If not, LUP decomposition is attempted. Then, the methods implementing the forward and backward substitution algorithms are called in succession.

LUP decomposition

Listing 8-6 shows the methods of the class `PMLUPDecomposition` implementing LUP decomposition.

To solve the system of equations 8.22 using LUP decomposition, one needs to write to evaluate the following expression:

```
(PMLUPDecomposition equations: #( (3 2 4) (2 -5 -1) (1 -2 2)) )
            solve: #(16 6 10).
```

This expression has been written on two lines to delineate the various steps. The first line creates an instance of the class `PMLUPDecomposition` by giving the coefficients of the equations, rows by rows. The second line is a call to the method `solve:` retrieving the solution of the system for the supplied constant vector.

Solving the same system for several constant vectors requires storing the LUP decomposition in a variable:

```
| s sol1 sol2 |
s := PMLUPDecomposition equations: #( (3 2 4) (2 -5 -1) (1 -2 2)).
sol1 := s solve: #(16 6 10).
sol2 := s solve: #(7 10 9).
```

When the first solution is fetched, the LUP decomposition is performed; then the solution is computed using forward and backward substitutions. When the second solution is fetched, only forward and backward substitutions are performed.

The proper creation class method, `equations:`, takes an array of arrays, the components of the matrix **A**. When a new instance is initialized the supplied coefficients are copied into the instance variable `rows` and the parity of the permutation is set to one. Copying the coefficients is necessary since the storage is reused during the decomposition steps.

A second creation method `direct:` allows the creation of an instance using the supplied system's coefficients directly. The user of this creation method must keep in mind that the coefficients are destroyed. This creation method can be used when the coefficients have been computed to the sole purpose of

solving the system of equations (*c.f.* sections 10.9 and 10.10 for an example of use).

The method `protectedDecomposition` handles the case when the matrix is singular by trapping the exception occurring in the method `decompose` performing the actual decomposition. When this occurs, the instance variable `permutation` is set to the integer 0 to flag the singular case. Then, any subsequent calls to the method `solve:` returns `nil`.

Listing 8-6 Implementation of the LUP decomposition

```
Object subclass: #PMLUPDecomposition
   instanceVariableNames: 'rows permutation parity'
   classVariableNames: ''
   package: 'Math-Matrix'
```

```
PMLUPDecomposition class >> direct: anArrayOfArrays
    ^ self new initialize: anArrayOfArrays
```

```
PMLUPDecomposition class >>  equations: anArrayOfArrays
        ^self new initialize: ( anArrayOfArrays collect: [ :each |
                                                    each deepCopy])
```

```
PMLUPDecomposition >> backwardSubstitution: anArray
    | n sum answer|
    n := rows size.
    answer := DhbVector new: n.
    n to: 1 by: -1 do:
        [ :i |
          sum := anArray at: i.
          ( i + 1) to: n do: [ :j | sum := sum - ( ( ( rows at: i)
                                          at: j) * ( answer at:
    j))].
          answer at: i put: sum / ( ( rows at: i) at: i) ].
    ^ answer
```

```
PMLUPDecomposition >> decompose
    | n |
    n := rows size.
    permutation := (1 to: n) asArray.
    1 to: ( n - 1) do:
        [ :k |
          self swapRow: k withRow: ( self largestPivotFrom: k);
               pivotAt: k ]
```

```
PMLUPDecomposition >> forwardSubstitution: anArray
    | n sum answer |
    answer := permutation collect: [ :each | anArray at: each].
    n := rows size.
    2 to: n do:
        [ :i |
          sum := answer at: i.
          1 to: ( i - 1) do: [ :j | sum := sum - ( ( ( rows at: i)
```

```
                                            at: j) * ( answer at:
    j))].
          answer at: i put: sum ].
    ^ answer
```

```
PMLUPDecomposition >> initialize: anArrayOfArrays
    rows := anArrayOfArrays.
    parity := 1
```

```
PMLUPDecomposition >> largestPivotFrom: anInteger
    | valueOfMaximum indexOfMaximum value |
    valueOfMaximum := ( ( rows at: anInteger) at: anInteger) abs.
    indexOfMaximum := anInteger.
    ( anInteger + 1) to: rows size do:
        [ :n |
          value := ( ( rows at: n) at: anInteger) abs.
          value > valueOfMaximum
                ifTrue: [ valueOfMaximum := value.
                          indexOfMaximum := n ] ].
    ^ indexOfMaximum
```

```
PMLUPDecomposition >> pivotAt: anInteger
    | inversePivot size k |
    inversePivot := 1 / ((rows at: anInteger) at: anInteger).
    size := rows size.
    k := anInteger + 1.
    k to: size
        do: [ :i |
              ( rows at: i) at: anInteger put: (( rows at: i) at:
                                                anInteger) *
    inversePivot.
              k to: size
                do: [ :j |
                      ( rows at: i) at: j put: ( ( rows at: i) at: j)
   - ( (( rows at: i) at: anInteger) * (( rows at: anInteger) at:
    j)) ]
            ]
```

```
PMLUPDecomposition >> printOn: aStream
    | first delimitingString n k |
    n := rows size.
    first := true.
    rows do:
        [ :row |
          first ifTrue: [ first := false ]
                ifFalse: [ aStream cr ].
          delimitingString := '('.
          row do:
            [ :each |
                aStream nextPutAll: delimitingString.
                each printOn: aStream.
```

```
                delimitingString := ' ' ].
          aStream nextPut: $) ]
```

```
PMLUPDecomposition >> protectedDecomposition
    [ self decompose] when: ExAll do: [ :signal | permutation := 0.
                                                  signal exitWith: nil
    ]
```

```
PMLUPDecomposition >> solve: anArrayOrVector
    permutation isNil
        ifTrue: [ self protectedDecomposition ].
    ^permutation = 0
        ifTrue: [ nil ]
        ifFalse:[ self backwardSubstitution: ( self
                                forwardSubstitution:
    anArrayOrVector)]
```

```
PMLUPDecomposition >> swapRow: anInteger1 withRow: anInteger2
    anInteger1 = anInteger2
        ifFalse: [ | swappedRow |
                   swappedRow := rows at: anInteger1.
                   rows at: anInteger1 put: ( rows at: anInteger2).
                   rows at: anInteger2 put: swappedRow.
                   swappedRow := permutation at: anInteger1.
                   permutation at: anInteger1 put: ( permutation at:

    anInteger2).
                   permutation at: anInteger2 put: swappedRow.
                   parity := parity negated.
                 ]
```

8.4 Computing the determinant of a matrix

The determinant of a matrix is defined as

$$\det \mathbf{A} = \begin{vmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{vmatrix} = \sum_{\pi} \operatorname{sign}(\pi)\, a_{1\pi(1)} a_{2\pi(1)} \cdots a_{n\pi(1)}, \tag{8.40}$$

where π represents a permutation of the indices 1 to n. The sum of equation 8.40 is made over all possible permutations. Thus, the sum contains $n!$ terms. Needless to say, the direct implementation of equation 8.40 to compute a determinant is highly inefficient.

Fortunately, the determinant of a matrix can be computed directly from its LUP decomposition. This comes from the following three properties of the determinants:

1. the determinant of a product of two matrices is the product of the determinants of the two matrices;
2. the determinant of a permutation matrix is 1 or -1 depending on the parity of the permutation;
3. the determinant of a triangular matrix is the product of its diagonal elements.

Applying the three properties above to equation 8.32 shows us that the determinant of the matrix **A** is simply the product of the diagonal elements of the matrix **U** times the sign of the permutation matrix **P**.

The parity of the permutation can be tracked while performing the LUP decomposition itself. The cost of this is negligible compared to the rest of the algorithm so that it can be done whether or not the determinant is needed . The initial parity is 1. Each additional permutation of the rows multiplies the parity by -1.

Our implementation uses the fact that objects of the class `Matrix` have an instance variable in which the LUP decomposition is kept. This variable is initialized using lazy initialization: if no LUP decomposition exists, it is calculated. Then the computation of the determinant is delegated to the LUP decomposition object.

Since the LUP decomposition matrices are kept within a single storage unit, a matrix, the LUP decomposition object calculates the determinant by taking the product of the diagonal elements of the matrix of the LUP decomposition object and multiplies the product by the parity of the permutation to obtain the final result.

Listing 8-7 shows the methods of classes `PMMatrix` and `PMLUPDecomposition` needed to compute a matrix determinant.

Listing 8-7 Methods to compute a matrix determinant

```
PMMatrix >> determinant
  ^self lupDecomposition determinant
```

```
PMLUPDecomposition >> determinant
    | n |
    permutation isNil
        ifTrue: [ self protectedDecomposition ].
    permutation = 0
        ifTrue: [ ^0 ]
```

8.5 Matrix inversion

The inverse of a square matrix **A** is denoted $\mathbf{A}^{-1}$. It is defined by the following equation:

$$\mathbf{A} \cdot \mathbf{A}^{-1} = \mathbf{I}, \tag{8.41}$$

where **I** is the identity matrix.

To determine the coefficients of the inverse matrix, one could use equation 8.41 as a system of n^2 linear equations if n is the dimension of the matrix **A**. this system could be solved using either Gaussian elimination (*c.f.* section 8.2) or LUP decomposition (*c.f.* section 8.3).

Using Gaussian elimination for such a system requires solving a system with n constant vectors. This could be done, but it is not very practical in terms of storage space except for matrices of small dimension. If we already have the LUP decomposition of the matrix **A**, however, this is indeed a solution. One can solve equation 8.41 for each columns of the matrix $\mathbf{A}^{-1}$. Specifically, $\mathbf{c}_i$, the i^{th} column of the matrix **A**, is the solution of the following system:

$$\mathbf{A} \cdot \mathbf{c}_i = \mathbf{e}_i \quad \text{for } i = 1, \ldots, n, \tag{8.42}$$

where $\mathbf{e}_i$ is the i^{th} column of the identity matrix, that is a vector with zero components except for the i^{th} component whose value is 1.

For large matrices, however, using LUP decomposition becomes quite slow. A cleverer algorithm for symmetric matrices is given in [**?**] with no name. In this book, we shall refer to this algorithm as the CLR algorithm (acronym of the authors of [**?**]).

Let **A** be a symmetric matrix. In section 8.1 we have seen that it can be written in the form:

$$\mathbf{A} = \begin{pmatrix} \mathbf{B} & \mathbf{C}^{\mathrm{T}} \\ \mathbf{C} & \mathbf{D} \end{pmatrix}, \tag{8.43}$$

where **B** and **D** are two symmetric matrices and **C** is in general not a square matrix. Then the inverse can be written as:

$$\mathbf{A}^{-1} = \begin{pmatrix} \mathbf{B}^{-1} + \mathbf{B}^{-1} \cdot \mathbf{C}^{\mathrm{T}} \cdot \mathbf{S}^{-1} \cdot \mathbf{C} \cdot \mathbf{B}^{-1} & \mathbf{B}^{-1} \cdot \mathbf{C}^{\mathrm{T}} \cdot \mathbf{S}^{-1} \\ -\mathbf{S}^{-1} \cdot \mathbf{C} \cdot \mathbf{B}^{-1} & \mathbf{S}^{-1} \end{pmatrix}, \tag{8.44}$$

where the matrix **S** is called the Schur's complement of the matrix **A** with respect to the partition of equation 8.43. The matrix **S** is also a symmetric matrix, given by the expression:

$$\mathbf{S} = \mathbf{D} - \mathbf{C} \cdot \mathbf{B}^{-1} \cdot \mathbf{C}^{\mathrm{T}}. \tag{8.45}$$

The reader can verify that equation 8.44 gives indeed the inverse of **A** by plugging 8.45 into 8.44 and carrying the multiplication with 8.43 in the conventional way. The result of the multiplication is an identity matrix. In particular, the result is independent of the type of partition described in equation 8.43.

The CRL algorithm consists of computing the inverse of a symmetric matrix using equations 8.43, 8.44 and 8.45 recursively. It is a divide-and-conquer algorithm in which the partition of equation 8.43 is further applied to the ma-

trices **B** and **S**. First the initial matrix is divided into four parts of approximately the same size. At each step the two inverses, $\mathbf{B}^{-1}$ and $\mathbf{S}^{-1}$ are computed using a new partition until the matrices to be inverted are either 2 by 2 or 1 by 1 matrices.

In the book of Cormen *et al.* [**?**] the divide and conquer algorithm supposes that the dimension of the matrix is a power of two to be able to use Strassen's algorithm for matrix multiplication. We have investigated an implementation of Strassen's algorithm, unfortunately its performance is still inferior to that of regular multiplication for matrices of dimension up to 512, probably because of the impact of garbage collection[8]. Indeed, the increase in memory requirement can be significant for matrices of moderate size. A 600 by 600 matrix requires 2.7 megabytes of storage. Extending it to a 1024 by 1024 would require 8 megabytes of storage.

Implementation strategy

In our implementation, we have generalized the divide and conquer strategy to any dimension. The dimension of the upper partition is selected at each partition to be the largest power of two smaller than the dimension of the matrix to be partitioned. Although the dimension of the matrices is not an integral power of two the number of necessary partitions remains a logarithmic function of the dimension of the original matrix, thus preserving the original idea of the CRL algorithm. It turns out that the number of necessary partitions is, in most cases, smaller than the number of partitions needed if the dimension of the original matrix is extended to the nearest largest power of two.

Both LUP and CRL algorithm perform within a time $\mathcal{O}(n^2)$ where n is the dimension of the matrix. Figure 8-8 shows the time needed to inverse a non-symmetrical matrix using CRL algorithm (solid line) and LUP decomposition (broken line), as well as the ratio between the two times (dotted line). The CRL algorithm has a large overhead but a smaller factor for the dependency on dimension. Thus, computing the inverse of a matrix using LUP decomposition is faster than the CLR algorithm for small matrices and slower for large matrices. As a consequence, our implementation of matrix inversion uses a different algorithm depending on the dimension of the matrix: if the dimension of the matrix is below a critical dimension, LUP decomposition is used; otherwise the CRL algorithm is used. In addition, LUP decomposition is always used if it has already been computed for another purpose.

On figure 8-8 we can determine that the critical dimension, below which the LUP decomposition works faster than the CRL algorithm, is about 36. These data were collected on a Pentium II running Windows NT 4.0. As this value is depending on the performance of the operating system, the reader is ad-

[8] The author thanks Thomas Cormen for enlightening E-mails on this subject.

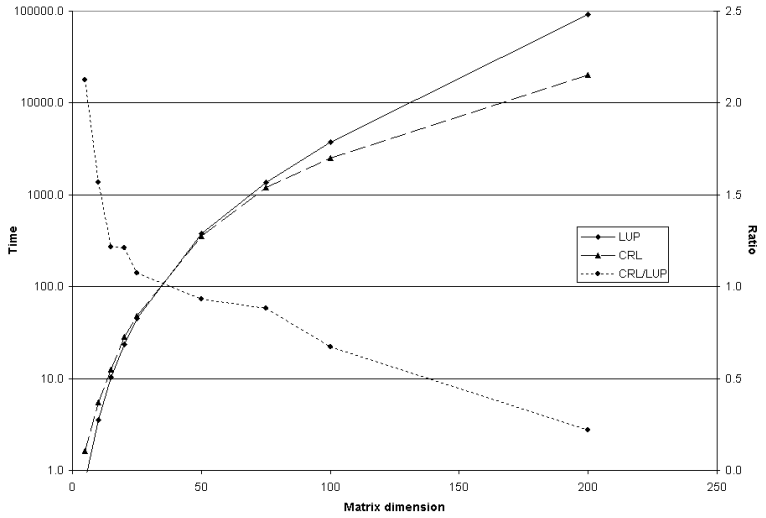


Figure 8-8 Comparison of inversion time for non-symmetrical matrices

vised to determine the critical dimension again when installing the classes on another system.

In practice, the CLR algorithm described in equations 8.43 to 8.45 can only be applied to symmetric matrices. In [**?**] Cormen *et al.* propose to generalize it to matrices of any size by observing the following identity:

$$\mathbf{A} \cdot \left[\left(\mathbf{A}^{\mathrm{T}} \cdot \mathbf{A} \right)^{-1} \cdot \mathbf{A}^{\mathrm{T}} \right] = \mathbf{I} \tag{8.46}$$

which can be verified for any matrix **A**. Thus, the expression in bracket can be considered as the inverse of the matrix **A**. In mathematics, it is called the pseudo-inverse or the Moore-Penrose inverse. Since the product $\mathbf{A}^{\mathrm{T}} \cdot \mathbf{A}$ is always a symmetric matrix, its inverse can be computed with the CRL algorithm. In practice, however. this technique is plagued with rounding errors and should be used with caution (*c.f.* section 8.5).

Matrix inversion implementation

Listing 8-9 shows the complete implementation in Pharo. It contains additional methods for the classes `PMMatrix` and `PMSymmetricMatrix`.

For symmetric matrices the method `inverse` first tests whether the dimension of the matrix is below a given threshold — defined by the class method `lupCRLCriticalDimension` — or whether the LUP decomposition of the matrix was already performed. In that case, the inverse is computed from the LUP decomposition using the method described at the beginning of section

8.5. Otherwise the CRL algorithm is used. The implementation of the CRL algorithm is straightforward thanks to the matrix operators defined in section 8.1.

For non-symmetric matrices the method `inverse` first tests whether the matrix is square or not. If the matrix is square, LUP decomposition is used. If the matrix is not square the pseudo inverse is computed using equation 8.46.

In both cases there is no error handling. Inverting a singular matrix produces an arithmetic error which must be handled by the calling method.

Listing 8-9 Implementation of matrix inversion

```
PMSymmetricMatrix class >> join: anArrayOfMatrices
    | rows n |
    rows := OrderedCollection new.
    n := 0.
    ( anArrayOfMatrices at: 1) rowsDo:
        [ :each |
          n := n + 1.
          rows add: each, ((anArrayOfMatrices at: 3) columnAt: n) ].
    n := 0.
    ( anArrayOfMatrices at: 2) rowsDo:
        [ :each |
          n := n + 1.
          rows add: ((anArrayOfMatrices at: 3) rowAt: n), each ].
    ^ self rows: rows
```

```
PMSymmetricMatrix class >> lupCRLCriticalDimension
    ^ 36
```

```
PMSymmetricMatrix >> crlInverse
    | matrices b1 cb1ct cb1 |
    matrices := self split.
    b1 := (matrices at: 1) inverse.
    cb1 := (matrices at: 3) * b1.
    cb1ct := (cb1 productWithTransposeMatrix: (matrices at: 3))
                asSymmetricMatrix.
    matrices at: 3 put: (matrices at: 2) * cb1.
    matrices at: 2 put: ((matrices at: 2) accumulateNegated: cb1ct)

    inverse.
    matrices at: 1 put: ( b1 accumulate: (cb1
                        transposeProductWithMatrix: (matrices at:
    3))).
    (matrices at: 3) negate.
    ^ self class join: matrices
```

```
PMSymmetricMatrix >> inverse
    ^ (rows size < self class lupCRLCriticalDimension or:
                                            [ lupDecomposition notNil
    ])
```

```
            ifTrue: [ self lupInverse ]
            ifFalse: [ self crlInverse ]
```

```
PMSymmetricMatrix >> split
    | n b d c |
    n := self largestPowerOf2SmallerThan: rows size.
    ^Array with: (self class rows: ((1 to: n) asVector collect: [
                              :k | (rows at: k) copyFrom: 1 to: n]))
             with:(self class rows: (((n+1) to: rows size)
      asVector collect: [ :k | (rows at: k) copyFrom: (n+1) to: rows
      size]))
             with: (self class superclass rows: (((n+1) to: rows
     size) asVector collect: [ :k | (rows at: k) copyFrom: 1 to: n]))
```

```
PMMatrix class >> lupCRLCriticalDimension
    ^ 40
```

```
PMMatrix >> inverse
    ^self isSquare
        ifTrue: [ self lupInverse ]
        ifFalse: [ self squared inverse * self transpose ]
```

```
PMMatrix >> largestPowerOf2SmallerThan: anInteger
    | m m2|
    m := 2.
    [ m2 := m * 2.
      m2 < anInteger ] whileTrue:[ m := m2 ].
    ^ m
```

```
PMMatrix >> lupInverse
    ^ self class rows: self lupDecomposition inverseMatrixComponents
```

Matrix inversion — Rounding problems

Operations with large matrices are well known to exhibit serious rounding problems. The reason is that the computation of the vector product of each row and column is a sum: the higher the dimension and the longer the sum. For large matrix dimensions the magnitude of the sum can mask small contributions from single products. Successive multiplications thus amplify initial small deviations. This is especially the case when computing the inverse of a general matrix using the CRL algorithm combined with the pseudo-inverse (8.46).

Now is the time to unveil the mystery example of section 1.3 about rounding errors propagation. The problem solved in this example is matrix inversion. The parameter describing the complexity of the problem is the dimension of the matrix. This is the quantity plotted along the x-axis of figure 1-1. Let $\mathbf{A}$ the matrix to be inverted. The matrix $\mathbf{M}$ defined by

$$\mathbf{M} = \mathbf{A}^{-1} \cdot \mathbf{A} - \mathbf{I}, \tag{8.47}$$

should have all its components equal to zero. The precision of the result is defined as the largest absolute value over all components of the matrix **M**. That quantity is plotted along the y-axis of figure 1-1.

Method A computes the inverse of the matrix using LUP decomposition, method B using the CRL algorithm. The *g*eneral data correspond to a matrix whose components were generated by a random number generator (*c.f.* section 9.4). They were all comprised between 0 and 1. For the *s*pecial data the matrix **A** is a covariance matrix (*c.f.* section 12.2) obtained by generating 1000 vectors with random components comprised between 0 and 1. For method B general data, the inverse of a non-symmetrical matrix is computed using the CRL algorithm combined with equation 8.46. In this general form, the CRL algorithm is faster the LUP for matrices of dimensions larger than about 165. The precision, however, is totally unreliable as can been seen on Figure 1-1.

8.6 Matrix eigenvalues and eigenvectors of a non-symmetric matrix

A non-zero vector **u** is called an *e*igenvector of the matrix **M** if there exists a complex number λ such that:

$$\mathbf{M} \cdot \mathbf{u} = \lambda \mathbf{u}. \tag{8.48}$$

the number λ is called an *eigenvalue* of the matrix **M**. Equation 8.48 implies that the matrix **M** must be a square matrix. In general a non-singular matrix of dimension n has n eigenvalues and eigenvectors. Some eigenvalues, however, may be double[9]. Discussing the existence of eigenvalues in the general case goes beyond the scope of this book. Equation 8.48 shows that an eigenvector is defined up to a constant[10]. One can prove that two eigenvectors of the same matrix, but corresponding to two different eigenvalues, are orthogonal to each other [**?**]. Thus, the eigenvectors of a matrix form a complete set of reference in a n dimensional space.

Computing the eigenvalues and eigenvectors of an arbitrary matrix is a difficult task. Solving this problem in the general case is quite demanding numerically. In the rest of this section we give an algorithm which works well when the absolute value of one of the eigenvalues is much larger than that of the others. The next section discusses Jacobi's algorithm finding all eigenvalues of a symmetrical matrix.

For an arbitrary square matrix the eigenvalue with the largest absolute value can be found with an iterative process. Let **u** be an arbitrary vector and let

[9] Eigenvalues are the roots of a polynomial of degree n. A double eigenvalue has two different eigenvectors.

[10] If the vector **u** is an eigenvector of the matrix **M** with eigenvalue λ, so are all vectors $\alpha\mathbf{u}$ for any $\alpha \neq 0$.

$\lambda_{\max}$ be the eigenvalue with the largest absolute value. Let us define the following series of vectors:

$$\begin{cases} \mathbf{u}_0 &= \mathbf{u}, \\ \mathbf{u}_k &= \frac{1}{\lambda_{\max}} \mathbf{M} \cdot \mathbf{u}_{k-1} \quad \text{for } k > 0. \end{cases} \tag{8.49}$$

It is easy to prove[11] that:

$$\lim_{k \to \infty} \mathbf{u}_k = \mathbf{u}_{\max}, \tag{8.50}$$

where $\mathbf{u}_{\max}$ is the eigenvector corresponding to $\lambda_{\max}$. Using this property, the following algorithm can be applied.

1. Set $\mathbf{u} = (1, 1, \ldots, 1)$.
2. Set $\mathbf{u}' = \mathbf{M}\mathbf{u}$.
3. Set $\lambda = u'_1$, that is the first component of the vector $\mathbf{u}'$.
4. Set $\mathbf{u} = \frac{1}{\lambda}\mathbf{u}'$.
5. Check for convergence of λ. Go to step 2 if convergence is not yet attained.

The algorithm will converge toward $\lambda_{\max}$ if the initial vector $\mathbf{u}$ is not an eigenvector corresponding to a null eigenvalue of the matrix $\mathbf{M}$. If that is the case, one can chose another initial vector.

Once the eigenvalue with the largest absolute value has been found, the remaining eigenvalues can be found by replacing the matrix $\mathbf{M}$ with the matrix:

$$\mathbf{M}' = \mathbf{M} \cdot \left(\mathbf{I} - \mathbf{u}_{\max} \otimes \mathbf{v}_{\max}\right), \tag{8.51}$$

where $\mathbf{I}$ is the identity matrix of same dimension as the matrix $\mathbf{M}$ and $\mathbf{v}_{\max}$ is the eigenvector of the matrix $\mathbf{M}^{\mathrm{T}}$ corresponding to $\lambda_{\max}$[12]. Using the fact that eigenvectors are orthogonal to each other, one can prove that the matrix $\mathbf{M}'$ of equation 8.51 has the same eigenvalues as the matrix except for $\lambda_{\max}$ which is replaced by 0. A complete proof of the above can be found in [**?**].

All eigenvalues and eigenvectors of the matrix $\mathbf{M}$ can be found by repeating the process above n times. However, this works well only if the absolute values of the eigenvalues differ from each consecutive ones by at least an order of magnitude. Otherwise, the convergence of the algorithm is not very good. In practice, this algorithm can only be used to find the first couple of eigenvalues.

[11] Hint: one must write the vector $\mathbf{u}$ as a linear combination of the eigenvectors of the matrix $\mathbf{M}$. Such linear combination exists because the eigenvectors of a matrix form a complete system of reference.

[12] The transpose of a matrix has the same eigenvalues, but not necessarily the same eigenvectors.

Finding the largest eigenvalue — General implementation

The object in charge of finding the largest eigenvalue is of course an instance of a subclass of the iterative process class described in 4.1. As the reader can see very few methods are required because most of the work is already implemented in the framework for iterative processes. The implementation is identical in both languages and will be discussed here. The largest eigenvalue finder has the following instance variables:

- `matrix` the matrix whose largest eigenvalue is sought,
- `eigenValue` the sought eigenvalue,
- `eigenVector` the sought eigenvector and
- `transposedEigenVector` the eigenvector of the transposed matrix.

The creation method takes the matrix as argument. Two accessor methods are supplied to retrieve the results, the eigenvalue and the eigenvector.

The method `initializeIterations` creates a vector to the matrix dimension and sets all its components equal to 1. As the algorithm progresses this vector will contain the eigenvector of the matrix. Similarly, a vector, which will contain the eigenvector of the transposed matrix, is created in the same way. In principle one should add a part to verify that this vector does not correspond to a null eigenvalue of the matrix. This small improvement is left as an exercise to the reader.

The algorithm is implemented within the single method `evaluateIteration` as described in section 4.1. The relative precision of the sought eigenvalue is the precision used to break out of the iterative process.

Since the algorithm determines both the eigenvalue and the eigenvector the object in charge of the algorithm keeps both of them and must give access to both of them. Two accessor methods are supplied to retrieve the results, the eigenvalue and the eigenvector.

The largest eigenvalue finder is responsible to create the object responsible for finding the next eigenvalue when needed. Thus, the eigenvector of the transposed matrix is also computed along with the regular eigenvector. The method `nextLargestEigenValueFinder` returns a new instance of the class, which can be used to compute the next largest eigenvalue, by computing a new matrix as described in equation 8.51.

Finding the largest eigenvalue implementation

Listing 8-10 shows the Pharo implementation of the class `PMLargestEigenValueFinder`, subclass of the class `PMIterativeProcess`.

The following code example shows how to use the class to find the first two largest eigenvalues of a matrix.

```
| m finder eigenvalue eigenvector nextFinder nextEigenvalue
    nextEigenvector |
m := PMMatrix rows: #((84 -79 58 55)
                      (-79 84 -55 -58)
                      (58 -55 84 79)
                      (55 -58 79 84)).
finder := PMLargestEigenValueFinder matrix: m.
eigenvalue := finder evaluate.
eigenvector := finder eigenvector.
nextFinder := finder nextLargestEigenValueFinder.
nextEigenvalue := nextFinder evaluate.
nextEigenvector := nextFinder eigenvector.
```

First the matrix `m` is defined from its components. Then, an instance of the class `PMLargestEigenValueFinder` is created for this matrix. The iterative process is started as described in section 4.1. Its result is the eigenvalue. The eigenvector is retrieved using an accessor method. Then, a new instance of `PMLargestEigenValueFinder` is obtained from the first one. The next largest eigenvalue and its eigenvector are retrieved from this new instance exactly as before.

Listing 8-10 Implementation of the search for the largest eigenvalue

```
PMIterativeProcess subclass: #PMLargestEigenValueFinder
   instanceVariableNames: 'matrix eigenvector transposeEigenvector'
   classVariableNames: ''
   package: 'Math-Matrix'
```

```
PMLargestEigenValueFinder class >> defaultMaximumIterations
    ^ 100
```

```
PMLargestEigenValueFinder class >> matrix: aMatrix
    ^ self new initialize: aMatrix; yourself
```

```
PMLargestEigenValueFinder >> matrix: aMatrix precision: aNumber
    ^ self new initialize: aMatrix;
              desiredPrecision: aNumber;
              yourself
```

```
PMLargestEigenValueFinder >> eigenvalue
    ^ result
```

```
PMLargestEigenValueFinder >> eigenvector
    ^ eigenvector * (1 / eigenvector norm)
```

```
PMLargestEigenValueFinder >> evaluateIteration
    | oldEigenvalue |
    oldEigenvalue := result.
    transposeEigenvector := transposeEigenvector * matrix.
    transposeEigenvector := transposeEigenvector
                * (1 / (transposeEigenvector at: 1)).
    eigenvector := matrix * eigenvector.
    result := eigenvector at: 1.
```

```
    eigenvector := eigenvector * (1 / result).
    ^oldEigenvalue isNil
        ifTrue: [ 2 * desiredPrecision]
        ifFalse: [ (result - oldEigenvalue) abs ]
```

```
PMLargestEigenValueFinder >> initialize: aMatrix
    matrix := aMatrix.
```

```
PMLargestEigenValueFinder >> initializeIterations
    eigenvector := PMVector new: matrix numberOfRows.
    eigenvector atAllPut: 1.0.
    transposeEigenvector := PMVector new: eigenvector size.
    transposeEigenvector atAllPut: 1.0
```

```
PMLargestEigenValueFinder >> nextLargestEigenValueFinder
    | norm |
    norm := 1 / (eigenvector * transposeEigenvector).
    ^self class
        new: matrix * ((PMSymmetricMatrix identity: eigenvector
                                                            size)
                        - (eigenvector * norm tensorProduct:

    transposeEigenvector))
        precision: desiredPrecision
```

8.7 Matrix eigenvalues and eigenvectors of a symmetric matrix

In the nineteen century Carl Jacobi discovered an efficient algorithm to find the eigenvalues of a symmetric matrix. Finding the eigenvalues of a symmetric matrix is easier since all eigenvalues are real.

In the section 8.6 we have mentioned that the eigenvectors of a matrix are orthogonal. Let $\mathbf{u}^{(1)}, \ldots, \mathbf{u}^{(n)}$ the set of eigenvectors of the matrix $\mathbf{M}$ such that $\mathbf{u}^{(i)} \cdot \mathbf{u}^{(i)} = 1$ for all i. Then, the matrix

$$\mathbf{O} = \begin{pmatrix} u_1^{(1)} & u_1^{(2)} & \ldots & u_1^{(n)} \\ u_2^{(1)} & u_2^{(2)} & \ldots & u_2^{(n)} \\ \vdots & \vdots & \ddots & \vdots \\ u_n^{(1)} & u_n^{(2)} & \ldots & u_n^{(n)} \end{pmatrix}, \tag{8.52}$$

where $u_i^{(k)}$ is the i^{th} component of the k^{th} eigenvector,is an orthogonal[13] matrix. That is, we have:

$$\mathbf{O}^{\mathrm{T}} \cdot \mathbf{O} = \mathbf{I}. \tag{8.53}$$

[13] An orthogonal matrix of dimension n is a rotation in the n-dimensional space.

Equation 8.53 is just another way of stating that the vectors $\mathbf{u}^{(1)}, \ldots, \mathbf{u}^{(n)}$ are orthogonal to each other and are all normalized to 1. Combining this property with the definition of an eigenvector (equation 8.48) yields:

$$\mathbf{O}^{\mathrm{T}} \cdot \mathbf{M} \cdot \mathbf{O} = \begin{pmatrix} \lambda_1 & 0 & \ldots & 0 \\ 0 & \lambda_2 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & \lambda_n \end{pmatrix}, \tag{8.54}$$

where $\lambda_1, \ldots, \lambda_n$ are the eigenvalues of the matrix $\mathbf{M}$.

The gist of Jacobi's algorithm is to apply a series of orthogonal transformations such that the resulting matrix is a diagonal matrix. It uses the fact that, for any orthogonal matrix $\mathbf{R}$, the matrix $\mathbf{R}^{\mathrm{T}}\mathbf{M} \cdot \mathbf{R}$ has the same eigenvalues as the matrix $\mathbf{M}$. This follows from the definition of an eigenvector (equation 8.48) and the property of an orthogonal matrix (equation 8.53).

An orthogonal matrix corresponds to a rotation of the system of reference axes. Each step of Jacobi's algorithm is to find an rotation, which annihilates one of the off-diagonal elements of the matrix resulting from that orthogonal transformation. Let $\mathbf{R}_1$ be such matrix and let us define

$$\mathbf{M}_1 = \mathbf{R}_1^{\mathrm{T}} \cdot \mathbf{M} \cdot \mathbf{R}_1. \tag{8.55}$$

Now, let us define the orthogonal transformation $\mathbf{R}_2$, which annihilates one of the off-diagonal elements of the matrix $\mathbf{M}_1$. The hope is that, after a certain number of steps m, the matrix

$$\begin{aligned} \mathbf{M}_m &= \mathbf{R}_m^{\mathrm{T}} \cdot \mathbf{M}_{m-1} \cdot \mathbf{R}_m \\ &= \mathbf{R}_m^{\mathrm{T}} \cdots \mathbf{R}_1^{\mathrm{T}} \cdot \mathbf{M} \cdot \mathbf{R}_1 \cdots \mathbf{R}_m \end{aligned} \tag{8.56}$$

becomes a diagonal matrix. Then the diagonal elements of the matrix $\mathbf{M}_m$ are the eigenvalues and the matrix

$$\mathbf{O}_m = \mathbf{R}_1 \cdots \mathbf{R}_m \tag{8.57}$$

is the matrix containing the eigenvectors.

Instead of annihilating just any diagonal element, one tries to annihiliate the element with the largest absolute value. This ensures the fastest possible convergence of the algorithm. Let m_{kl} be the off-diagonal element of the matrix $\mathbf{M}$ with the largest absolute value. We define a matrix $\mathbf{R}_1$ with compo-

nents:

$$\begin{cases} r_{kk}^{(1)} &= \cos\vartheta, \\ r_{ll}^{(1)} &= \cos\vartheta, \\ r_{kl}^{(1)} &= -\sin\vartheta, \\ r_{lk}^{(1)} &= \sin\vartheta, \\ r_{ii}^{(1)} &= 1 \quad \text{for } i \neq k, l, \\ r_{ij}^{(1)} &= 0 \quad \text{for } i \neq j, i \text{ and } j \neq k, l. \end{cases} \tag{8.58}$$

The reader can verify that the matrix $\mathbf{R}_1$ is an orthogonal matrix. The new matrix $\mathbf{M}_1 = \mathbf{R}_1^{\mathsf{T}} \cdot \mathbf{M} \cdot \mathbf{R}_1$ has the same components as the matrix $\mathbf{M}$ except for the rows and columns k and l. That is, we have

$$\begin{cases} m_{kk}^{(1)} &= \cos^2\vartheta m_{kk} + \sin^2\vartheta m_{ll} - 2\sin\vartheta\cos\vartheta m_{kl}, \\ m_{ll}^{(1)} &= \sin^2\vartheta m_{kk} + \cos^2\vartheta m_{ll} + 2\sin\vartheta\cos\vartheta m_{kl}, \\ m_{kl}^{(1)} &= \left(\cos^2\vartheta - \sin^2\vartheta\right) m_{kl} + \sin\vartheta\cos\vartheta\left(m_{kk} - m_{ll}\right), \\ m_{ik}^{(1)} &= \cos\vartheta m_{ik} - \sin\vartheta m_{il} \quad \text{for } i \neq k, l, \\ m_{il}^{(1)} &= \cos\vartheta m_{il} + \sin\vartheta m_{ik} \quad \text{for } i \neq k, l, \\ m_{ij}^{(1)} &= m_{ij} \quad \text{for } i \neq k, l \text{ and } j \neq k, l. \end{cases} \tag{8.59}$$

In particular, the angle of rotation can be selected such that $m_{kl}^{(1)} = 0$. That condition yields the following equation for the angle of the rotation:

$$\frac{\cos^2\vartheta - \sin^2\vartheta}{\sin\vartheta\cos\vartheta} = \frac{m_{ll} - m_{kk}}{m_{kl}} = \alpha, \tag{8.60}$$

where the constant α is defined by equation 8.60. Introducing the variable $t = \tan\vartheta$, equation 8.60 can be rewritten as:

$$t^2 + 2\alpha t - 1 = 0. \tag{8.61}$$

Since equation 8.61 is a second order equation, there are two solutions. To minimize rounding errors, it is preferable to select the solution corresponding to the smallest rotation[**?**]. The solution of equation 8.61 has already been discussed in section 1.3 for the case where α is positive. For any α, it can be written as:

$$t = \frac{\operatorname{sign}(\alpha)}{|\alpha| + \sqrt{\alpha^2 + 1}}. \tag{8.62}$$

In fact, the value of the angle ϑ does not need to be determined. We have:

$$\begin{cases} \cos\vartheta &= \dfrac{1}{\sqrt{t^2+1}}, \\ \sin\vartheta &= t\cos\vartheta. \end{cases} \tag{8.63}$$

Let us now introduce the quantities σ and τ defined as

$$\begin{cases} \sigma &= \sin\vartheta, \\ \tau &= \dfrac{\sin\vartheta}{1+\cos\vartheta}. \end{cases} \tag{8.64}$$

Then equations 8.59 can be rewritten as

$$\begin{cases} m_{kk}^{(1)} &= m_{kk} - t m_{kl}, \\ m_{ll}^{(1)} &= m_{ll} + t m_{kl}, \\ m_{kl}^{(1)} &= 0, \\ m_{ik}^{(1)} &= m_{ik} - \sigma\left(m_{il} + \tau m_{ik}\right) \quad \text{for } i \neq k, l, \\ m_{il}^{(1)} &= m_{il} + \sigma\left(m_{ik} - \tau m_{il}\right) \quad \text{for } i \neq k, l, \\ m_{ij}^{(1)} &= m_{ij} \quad \text{for } i \neq k, l \text{ and } j \neq k, l. \end{cases} \tag{8.65}$$

Finally, we must prove that the transformation above did not increase the absolute values of the remaining off-diagonal elements of the matrix $\mathbf{M}_1$. Using equations 8.59 the sum of the off-diagonal elements of the matrix $\mathbf{M}_1$ is:

$$\sum_{i \neq j} \left(m_{ij}^{(1)}\right)^2 = \sum_{i \neq j} m_{ij}^2 - 2m_{kl}^2. \tag{8.66}$$

Thus, this sum is always less that the sum of the squared off-diagonal elements of the matrix $\mathbf{M}$. In other words the algorithm will always converge.

Jacobi's algorithm

Now we have all the elements to implement Jacobi's algorithm. The steps are described hereafter:

1. Set the matrix $\mathbf{M}$ to the matrix whose eigenvalues are sought.
2. Set the matrix $\mathbf{O}$ to an identity matrix of the same dimension as the matrix $\mathbf{M}$.
3. Find the largest off-diagonal element, m_{kl}, of the matrix $\mathbf{M}$.
4. Build the orthogonal transformation $\mathbf{R}_1$ annihilating the element m_{kl}.

5. Build the matrix $\mathbf{M}_1 = \mathbf{R}_1^T \cdot \mathbf{M} \cdot \mathbf{R}_1$.
6. If $|m_{kl}|$ is less than the desired precision go to step 8.
7. Let $\mathbf{M} = \mathbf{M}_1$ and $\mathbf{O} = \mathbf{O} \cdot \mathbf{R}_1$; go to step 3.
8. The eigenvalues are the diagonal elements of the matrix $\mathbf{M}$ and the eigenvectors are the rows of the matrix $\mathbf{O}$.

Strictly speaking, Jacobi's algorithm should be stopped if the largest off-diagonal element of matrix $\mathbf{M}_1$ is less than the desired precision. However, equation 8.66 guaranties that the largest off-diagonal element of the matrix after each step of Jacobi's algorithm is always smaller that the largest off-diagonal element of the matrix before the step. Thus. the stopping criteria proposed above can safely be used. This slight overkill prevents us from scanning the off-diagonal elements twice per step.

As the algorithm converges, α becomes very large. As discussed in section 1.3, the solution of equation 8.61 can be approximated with

$$t \approx \frac{1}{2\alpha}. \tag{8.67}$$

This expression is used when the computation of α^2 causes an overflow while evaluating equation 8.62.

Jacobi's algorithm — Smalltalk implementation

Jacobi's algorithm is an iterative algorithm. The object implementing Jacobi's algorithm is an instance of the class `PMJacobiTransform`; it is a subclass of the iterative process discussed in section 4.1.

When an instance of the class `JacobiTransform` is created, the matrix whose eigenvalues are sought is copied into the matrix $\mathbf{M}$. This permits to use the same storage over the duration of the algorithm since equations 8.65 can be evaluated in place. Actually, only the upper half of the components needs to be stored since the matrix is a symmetric matrix.

The method `evaluateIteration` finds the largest off-diagonal element and performs the Jacobi step (equations 8.65) for that element. During the search for the largest off-diagonal element, the precision of the iterative process is set to the absolute value of the largest off-diagonal element. This is one example where it does not make sense to compute a relative precision. Actually, the precision returned by the method `evaluateIteration` is that of the previous iteration, but it does not really matter to make one iteration too much.

The method `finalizeIterations` performs a bubble sort to place the eigenvalues in decreasing order of absolute value. Bubble sorting is used instead of using a `SortedCollection` because one must also exchange the corresponding eigenvectors.

The result of the iterative process is an array containing the sorted eigenvalues plus the transformation matrix **O** containing the eigenvectors. Extracting these results is language dependent.

Listing 8-11 shows the implementation of Jacobi's algorithm.

The following code example shows how to use the class to find the eigenvalues and eigenvectors of a symmetric matrix.

```
| m jacobi eigenvalues eigenvectors |
m := PMSymmetricMatrix rows: #((84 -79 58 55)
                               (-79 84 -55 -58)
                               (58 -55 84 79)
                               (55 -58 79 84)).
jacobi := PMJacobiTransformation matrix: m.
eigenvalues := jacobi evaluate.
eigenvectors := jacobi transform columnsCollect: [ :each | each].
```

First the matrix `m` is defined from its components. Then, an instance of the class `PMJacobiTransformation` is created for this matrix. The iterative process is started as described in section 4.1. Its result is an array containing the eigenvalues sorted in decreasing order. The corresponding eigenvectors are retrieved from the columns of the matrix **O** obtained from the method `transform`.

The class `PMJacobiTransformation` has two instance variables

- `lowerRows` an array of array containing the lower part of the matrix and
- `transform` the components of the matrix **O**.

Since the matrix **M** is symmetric there is no need to keep all of its components. This not only reduces storage but also speeds up somewhat the algorithm because one only need to transform the lower part of the matrix.

The instance variable `result` contains the sorted eigenvalues at the end of the iterative process. The method `transform` returns the symmetric matrix **O** whose columns contain the eigenvectors in the same order. The code example shown at the beginning of this section shows how to obtain the eigenvectors from the matrix.

Listing 8-11 Implementation of Jacobi's algorithm

```
PMIterativeProcess subclass: #PMJacobiTransformation
   instanceVariableNames: 'lowerRows transform'
   classVariableNames: ''
   package: 'Math-Matrix'
```

```
PMJacobiTransformation class >> matrix: aSymmetricMatrix
    ^ super new initialize: aSymmetricMatrix
```

```
PMJacobiTransformation class >> new
    ^ self error: 'Illegal creation message for this class'
```

```
PMJacobiTransformation >> evaluateIteration
    | indices |
    indices := self largestOffDiagonalIndices.
    self transformAt: (indices at: 1) and: (indices at: 2).
    ^ precision
```

```
PMJacobiTransformation >> exchangeAt: anInteger
    | temp n |
    n := anInteger + 1.
    temp := result at: n.
    result at: n put: ( result at: anInteger).
    result at: anInteger put: temp.
    transform do:
        [ :each |
          temp := each at: n.
          each at: n put: ( each at: anInteger).
          each at: anInteger put: temp ].
```

```
PMJacobiTransformation >> finalizeIterations
    | n |
    n := 0.
    result := lowerRows collect:
                    [ :each |
                    n := n + 1.
                    each at: n ].
    self sortEigenValues
```

```
PMJacobiTransformation >> initialize: aSymmetricMatrix
    | n m |
    n := aSymmetricMatrix numberOfRows.
    lowerRows := Array new: n.
    transform := Array new: n.
    1 to: n do:
        [ :k |
          lowerRows at: k put: ( ( aSymmetricMatrix rowAt: k)
                                                    copyFrom: 1 to:
    k).
          transform at: k put: ( ( Array new: n) atAllPut: 0; at: k
                                                     put: 1;
    yourself) ].
    ^ self
```

```
PMJacobiTransformation >> largestOffDiagonalIndices
    | n m abs |
    n := 2.
    m := 1.
    precision := ( ( lowerRows at: n) at: m) abs.
    1 to: lowerRows size do:
        [ :i |
          1 to: ( i - 1) do:
            [ :j |
```

```
            abs := ( ( lowerRows at: i) at: j) abs.
            abs > precision
              ifTrue: [ n := i.
                        m := j.
                        precision := abs ] ] ].
    ^ Array with: m with: n
```

```
PMJacobiTransformation >> printOn: aStream
    | first |
    first := true.
    lowerRows do:
        [ :each |
          first ifTrue: [ first := false ]
                ifFalse: [ aStream cr ].
          each printOn: aStream ].
```

```
PMJacobiTransformation >> sortEigenValues
    | n bound m |
    n := lowerRows size.
    bound := n.
    [ bound = 0 ]
        whileFalse: [ m := 0.
                      1 to: bound - 1 do:
                        [ :j |
                          (result at: j) abs > (result at: j + 1) abs
                            ifFalse: [ self exchangeAt: j.
                                       m := j ] ].
                        bound := m ].
```

```
PMJacobiTransformation >> transform
    ^ PMMatrix rows: transform
```

```
PMJacobiTransformation >> transformAt: anInteger1 and: anInteger2
    | d t s c tau apq app aqq arp arq |
    apq := ( lowerRows at: anInteger2) at: anInteger1.
    apq = 0
        ifTrue: [ ^ nil ].
    app := (lowerRows at: anInteger1) at: anInteger1.
    aqq := (lowerRows at: anInteger2) at: anInteger2.
    d := aqq - app.
    arp := d * 0.5 / apq.
    t := arp > 0
        ifTrue: [ 1 / ( ( arp squared + 1) sqrt + arp)]
        ifFalse:[ 1 / ( arp - ( arp squared + 1) sqrt)].
    c := 1 / ( t squared + 1) sqrt.
    s := t * c.
    tau := s / ( 1 + c).
    1 to: (anInteger1 - 1)
        do: [ :r |
              arp := (lowerRows at: anInteger1) at: r.
              arq := (lowerRows at: anInteger2) at: r.
```

```
            (lowerRows at: anInteger1) at: r put: ( arp - ( s *
                                                    (tau * arp +
    arq))).
            (lowerRows at: anInteger2) at: r put: ( arq + ( s *
                                                  (arp - (tau *
    arq)))).
          ].
    ( anInteger1 + 1) to: ( anInteger2 - 1)
        do: [ :r |
            arp := (lowerRows at: r) at: anInteger1.
            arq := (lowerRows at: anInteger2) at: r.
            (lowerRows at: r) at: anInteger1 put: ( arp - ( s *
                                                    (tau * arp +
    arq))).
            (lowerRows at: anInteger2) at: r put: ( arq + ( s *
                                                  (arp - (tau *
    arq)))).
          ].
    ( anInteger2 + 1) to: lowerRows size
        do: [ :r |
            arp := ( lowerRows at: r) at: anInteger1.
            arq := ( lowerRows at: r) at: anInteger2.
            (lowerRows at: r) at: anInteger1 put: ( arp - ( s *
                                                    (tau * arp +
    arq))).
            (lowerRows at: r) at: anInteger2 put: ( arq + ( s *
                                                  (arp - (tau *
    arq)))).
          ].
    1 to: lowerRows size
        do: [ :r |
            arp := ( transform at: r) at: anInteger1.
            arq := ( transform at: r) at: anInteger2.
            (transform at: r) at: anInteger1 put: ( arp - ( s *
                                                    (tau * arp +
    arq))).
            (transform at: r) at: anInteger2 put: ( arq + ( s *
                                                  (arp - (tau *
    arq)))).
          ].
    (lowerRows at: anInteger1) at: anInteger1 put: ( app - (t *

    apq)).
    (lowerRows at: anInteger2) at: anInteger2 put: ( aqq + (t *

    apq)).
    (lowerRows at: anInteger2) at: anInteger1 put: 0.
```

CHAPTER 9

Elements of statistics

La statistique est la première des sciences inexactes.[1]
Edmond et Jules de Goncourt

Statistical analysis comes into play when dealing with a large amount of data. Obtaining information from the statistical analysis of data is the subject of chapter 10. Some sections of chapter 12 are also using statistics. Concepts needed by statistics are based on probability theory.

This chapter makes a quick overview of the concepts of probability theory. It is the third (and last) chapter of this book where most of the material is not useful *per* se. Figure 9-1 shows the classes described in this chapter. All these classes, however, are used extensively in the remaining chapters of this book. The example on how to use the code are kept to a minimum since real examples of use can be found in the next chapters.

An in-depth description of probability theory is beyond the scope of this book. The reader in the need for additional should consult the numerous textbooks on the subject, [**?**] or [**?**] for example.

9.1 Statistical moments

When one measures the values of an observable random variable, each measurement gives a different magnitude. Assuming measurement errors are negligible, the fluctuation of the measured values correspond to the distribution of the random variable. The problem to be solved by the experimenter is to determine the parameters of the distribution from the observed values.

[1] Statistics is the first of the inexact sciences.

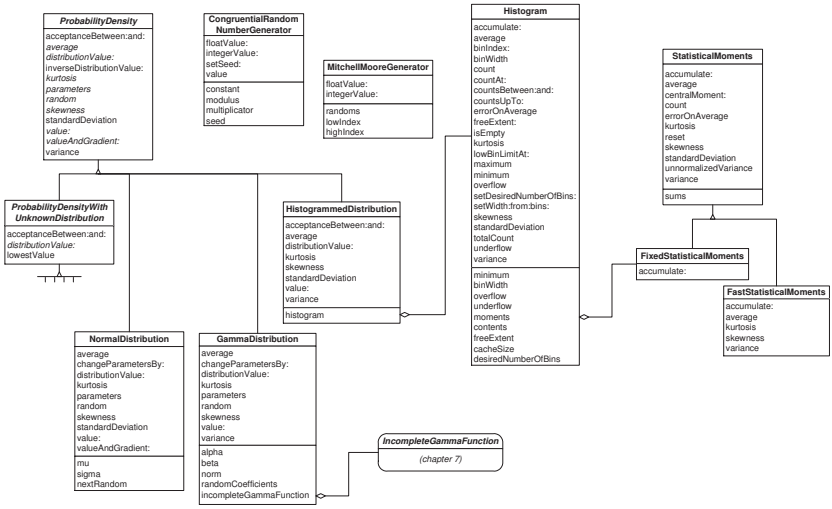


Figure 9-1 Classes related to statistics

Statistical moments can contribute to the characterization of the distribution[2].

Given a set of measurements, $x_1, \ldots, x_n$, of the values measured for a random variable one defines the moment of k order by:

$$M_k = \frac{1}{n} \sum_{i=1}^{n} x_i^k. \tag{9.1}$$

In particular, the moment of first order is the mean or average of the set of data:

$$\bar{x} = M_1 = \frac{1}{n} \sum_{i=1}^{n} x_i. \tag{9.2}$$

The central moments of $k^{\mathbf{th}}$ order is defined by:

$$m_k = \frac{1}{n} \sum_{i=1}^{n} (x_i - \bar{x})^k . \tag{9.3}$$

where k is larger than 1. The central moments are easily expressed in terms of the moments. We have:

$$m_k = \sum_{j=0}^{k} \binom{k}{j} (-\bar{x})^{k-j} M_j, \tag{9.4}$$

[2]Central moments are related to the coefficients of the Taylor expansion of the Fourier transform of the distribution function.

where $\binom{k}{j}$ are the binomial coefficients.

Some statistical parameters are defined on the central moments. The variance of a set of measurement is the central moment of second order. The standard deviation, s, is the square root of the variance given by the following formula:

$$s^2 = \frac{n}{n-1} m_2 = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2 . \tag{9.5}$$

The factor in front the central moment of second order is called Bessel's correction factor. This factor removes the bias of the estimation when the standard deviation is evaluated over a finite sample. The standard deviation measures the spread of the data around the average.

Many people believe that the standard deviation is the error of the average. This is not true: the standard deviation describes how much the data are spread around the average. It thus represents the error of a single measurement. An estimation of the standard deviation of the average value is given by the following formula:

$$s_{\bar{x}}^2 = \frac{s^2}{n} \quad \text{or} \quad s_{\bar{x}} = \frac{s}{\sqrt{n}}. \tag{9.6}$$

This expression must be taken as the error on the average when performing a least square fit on averaged data, for example.

Two quantities are related to the central moments of $30.44rd$ and 4 order. Each of these quantities are normalized by the adequate power of the standard deviation needed to yield a quantity without dimension.

The skewness is defined by:

$$a = \frac{n}{(n-1)(n-2)s^3} m_3 = \frac{1}{(n-1)(n-2)} \sum_{i=1}^{n} \left(\frac{x_i - \bar{x}}{s} \right)^3 . \tag{9.7}$$

The skewness is a measure of the asymmetry of a distribution. If the skewness is positive, the observed distribution is asymmetric toward large values and vice-versa.

The kurtosis is defined by

$$\begin{aligned} k = & \frac{n(n+1)}{(n-1)(n-2)(n-3)s^4} m_4 - \frac{3(n-1)^2}{(n-2)(n-3)} \\ = & \frac{(n+1)}{(n-1)(n-2)(n-3)} \sum_{i=1}^{n} \left(\frac{x_i - \bar{x}}{s} \right)^4 - \frac{3(n-1)^2}{(n-2)(n-3)} \end{aligned} \tag{9.8}$$

The kurtosis is a measure of the peakedness or flatness of a distribution in the region of the average. The subtracted term in equation 9.8 is a convention defining the kurtosis of the normal distribution as 0[3].

As we have seen, the average, standard deviation, skewness and kurtosis are parameters, which helps characterizing a distribution of observed values. To keep track of these parameters, it is handy to define an object whose responsibility is to accumulate the moments up to order 4. One can then easily compute the parameters of the distribution. It can be used in all cases where distribution parameters are needed.

Statistical moments — implementation

To describe this implementation we must anticipated on the next section: the class **P**MFastStatisticalMoments implementing statistical moments as described in Section 9.1 is a subclass of the class defined in section 9.2.

Space allocation is handled by the superclass. The class PMFastStatisticalMoments uses this allocation to store the moments (instead of the central moments). The method accumulate: perform the accumulation of the moments. The methods average, variance, skewness and kurtosis compute the respective quantities using explicit expansion of the central moments as a function of the moments.

The computation of the standard deviation and of the error on the average are handled by the superclass (*c.f.* listing **??**).

Listing 9-2 shows the Smalltalk implementation. The class PMFastStatisticalMoments is a subclass of class PMStatisticalMoments presented in listing 9-3 of section 9.2. The reason for the split into two classes will become clear in section 9.2.

The following code shows how to use the class PMFastStatisticalMoments to accumulate measurements of a random variable and to extract the various distribution parameters discussed in section 9.1.

```
| accumulator valueStream average stdev skewness kurtosis |
accumulator := PMFastStatisticalMoments new.
[ valueStream atEnd ]
        whileFalse: [ accumulator accumulate: valueStream next ].
average := accumulator average.
stdev := accumulator standardDeviation.
skewness := accumulator skewness.
kurtosis := accumulator kurtosis
```

[3] One talks about a *p*latykurtic distribution when the kurtosis is negative, that is the peak of the distribution is flatter than that of the normal distribution. Student (*c.f.* section 10.2) and Cauchy (*c.f.* section **??**) distributions are platykurtic. The opposite is called *l*eptokurtic. The Laplace (*c.f.* section **??**) distribution is leptokurtic.

This example assumes that the measurement of the random variable are obtained from a stream. The exact implementation of the stream is not shown here.

After the declarative statements, the first executable statement creates a new instance of the class PMFastStatisticalMoments with the default dimension. This default allocates enough storage to accumulate up to the moment of 4 order. The next two lines are the accumulation proper using a whileFalse: construct and the general behavior of a stream. The last four lines extract the main parameters of the distribution.

If any of the distribution's parameters — average, variance, skewness or kurtosis — cannot be computed, we use self shouldNoImplement.

Listing 9-2 Smalltalk fast implementation of statistical moments

```
PMStatisticalMoments subclass: #PMFastStatisticalMoments
   instanceVariableNames: ''
   classVariableNames: ''
   package: 'Math-StatisticalMoments'
```

```
PMFastStatisticalMoments >> accumulate: aNumber
    | var |
    var := 1.
    1 to: moments size
        do:
            [ :n |
            moments at: n put: (moments at: n) + var.
            var := var * aNumber ]
```

```
PMFastStatisticalMoments >> average
    self count = 0 ifTrue: [ ^nil ].
    ^ (moments at: 2) / self count
```

```
PMFastStatisticalMoments >> kurtosis
    | var x1 x2 x3 x4 kFact kConst n m4 xSquared |
    n := self count.
    n < 4 ifTrue: [^nil].
    var := self variance.
    var = 0 ifTrue: [^nil].
    x1 := (moments at: 2) / n.
    x2 := (moments at: 3) / n.
    x3 := (moments at: 4) / n.
    x4 := (moments at: 5) / n.
    xSquared := x1 squared.
    m4 := x4 - (4 * x1 * x3) + (6 * x2 * xSquared) - (xSquared
                                                        squared *
    3).
    kFact := n * (n + 1) / (n - 1) / (n - 2) / (n - 3).
    kConst := 3 * (n - 1) * (n - 1) / (n - 2) / (n - 3).
    ^ kFact * (m4 * n / var squared) - kConst
```

```
PMFastStatisticalMoments >> skewness
    | x1 x2 x3 n stdev |
    n := self count.
    n < 3 ifTrue: [ ^nil ].
    stdev := self standardDeviation.
    stdev = 0 ifTrue: [^nil].
    x1 := (moments at: 2) / n.
    x2 := (moments at: 3) / n.
    x3 := (moments at: 4) / n.
    ^ (x3 - (3 * x1 * x2) + (2 * x1 * x1 * x1)) * n * n
        / (stdev squared * stdev * (n - 1) * (n - 2))
```

```
PMFastStatisticalMoments >> variance
    | n |
    n := self count.
    n < 2 ifTrue: [^nil].
    ^ ((moments at: 3) - ((moments at: 2) squared / n)) / (n - 1)
```

9.2 Robust implementation of statistical moments

The methods used to implement the computation of the central moments in the previous section is prone to rounding errors. Indeed, contribution from values distant from the average can totally offset a result, however infrequent they are. Such an effect is worse when the central moments are derived from the moments. This section gives an algorithm ensuring minimal rounding errors.

The definition of statistical moments is based on the concept of expectation value. The expectation value is a linear operator over all functions of the random variable. If one measures the values of the random variable n times, the expectation value of a function $f(x)$ of a random variable x is estimated by the following expression:

$$\langle f(x)\rangle_n = \frac{1}{n}\sum_{i=1}^{n} f(x_i), \tag{9.9}$$

where the values $x_1, \ldots, x_n$ are the measurements of the random variable. A comparison of equation 9.9 with 9.2 shows that the average is simply the expectation value of the function $f(x) = x$. The central moment of order k is the expectation value of the function $(x - \bar{x})^k$:

$$\left\langle (x-\bar{x})^k \right\rangle_n = \frac{1}{n}\sum_{i=1}^{n} (x_i - \bar{x})^k. \tag{9.10}$$

To miminize rounding errors, one computes the changes occurring to the central moments when a new value is taken into account. In other words, one computes the value of a central moment over $n+1$ values as a function of

the central moment over n values and the $(n+1)$ value. For the average, we have

$$\begin{aligned}\langle x\rangle_{n+1} &= \frac{1}{n+1}\sum_{i=1}^{n+1} x_i \\ &= \frac{x_{n+1}}{n+1} + \frac{1}{n+1}\sum_{i=1}^{n} x_i \\ &= \frac{x_{n+1}}{n+1} + \frac{n}{n+1}\langle x\rangle_n \\ &= \frac{x_{n+1}}{n+1} + \left(1 - \frac{1}{n+1}\right)\langle x\rangle_n \\ &= \langle x\rangle_n - \frac{\langle x\rangle_n - x_{n+1}}{n+1}.\end{aligned} \tag{9.11}$$

Thus, the estimator of the average over $n+1$ measurements can be computed from the estimator of the average over n measurements by subtracting a small correction, Δ_{n+1}, given by:

Main equation⇒

$$\begin{aligned}\Delta_{n+1} &= \langle x\rangle_n - \langle x\rangle_{n+1} \\ &= \frac{\langle x\rangle_n - x_{n+1}}{n+1}.\end{aligned} \tag{9.12}$$

The expression in the numerator of equation 9.12 subtracts two quantities of comparable magnitude. This ensures a minimization of the rounding errors.

A similar derivation can be made for the central moments of higher orders. A complete derivation is given in appendix **??**. The final expression is

Main equation⇒

$$\left\langle (x-\bar{x})^k \right\rangle_{n+1} = \frac{n}{n+1}\left\{\left[1-(-n)^{k-1}\right]\Delta_{n+1}^k + \sum_{l=2}^{k}\binom{l}{k}\left\langle (x-\mu)^l\right\rangle_n \Delta_{n+1}^{k-l}\right\}. \tag{9.13}$$

The reader can verify the validity of equation 9.13 by verifying that it gives 1 for $k=0$ and 0 for $k=1$. Put in this form, the computation of the central moment estimators minimizes indeed rounding errors. For the central moment of order 2 we have:

Main equation⇒

$$\left\langle (x-\bar{x})^2 \right\rangle_{n+1} = \frac{n}{n+1}\left\{(1+n)\,\Delta_{n+1}^2 + \left\langle (x-\bar{x})^2\right\rangle_n\right\}. \tag{9.14}$$

For the central moment of order 3 we have:

Main equation⇒

$$\left\langle (x-\bar{x})^3 \right\rangle_{n+1} = \frac{n}{n+1}\left\{\left(1-n^2\right)\Delta_{n+1}^3 + 3\left\langle (x-\bar{x})^2\right\rangle_n \Delta_{n+1} + \left\langle (x-\bar{x})^3\right\rangle_n\right\}. \tag{9.15}$$

For the central moment of order 4 we have:

Main equation⇒

$$\begin{aligned}\left\langle (x-\bar{x})^4 \right\rangle_{n+1} = \frac{n}{n+1} &\left\{\left(1+n^3\right)\Delta_{n+1}^4 + 6\left\langle (x-\bar{x})^2\right\rangle_n \Delta_{n+1}^2\right. \\ &\left. +4\left\langle (x-\bar{x})^3\right\rangle_n \Delta_{n+1} + \left\langle (x-\bar{x})^4\right\rangle_n\right\}.\end{aligned} \tag{9.16}$$

Robust central moments — implementation

The class PMStatisticalMoments has a single instance variable moments used to store the accumulated central moments.

The evaluation of equation 9.13 is not as hard as it seems from a programming point of view. One must remember that the binomial coefficients can be obtained by recursion (Pascal triangle). Furthermore, the terms of the sum can be computed recursively from those of the previous order so that raising the correction Δ_{n+1} to an integer power is never made explicitly. Equation 9.13 is implemented in method accumulate. The reader will notice that the binomial coefficients are computed inside the loop computing the sum. Accumulating the central moments using equation 9.13 has the advantage that the estimated value of the central moment is always available. Nevertheless, accumulation is about 2 times slower than with the brute force method exposed in section 9.1. The reader must decide between speed and accuracy to chose between the two implementations.

The class PMFixedStatisticalMoments is a subclass of class PMStatisticalMoments specialized in the accumulation of central moments up to order 4. Instead of implementing the general equation 9.13, the central moments are accumulated using equations 9.14, 9.15 and 9.16. The only instance method redefined by this class is the method accumulate. All other computations are performed using the methods of the superclass.

Listing **??** shows the implementation of the robust statistical moments. Listing **??** shows a specialization to optimize the speed of accumulation for the most frequently used case (accumulation up to the 4 order).

Using the class is identical for all classes of the hierarchy. Thus, the code example presented in section 9.1 is also valid for these two classes.

The creation method new: takes as argument the highest order of the accumulated moments. The corresponding initialization method allocates the required storage. The creation method new corresponds to the most frequent usage: the highest order is 4.

The methods computing the distribution parameters — average, variance, skewness and kurtosis — are using the method centralMoment: retrieving the central moment of a given order. They will return nil if not enough data as been accumulated in the moments.

Listing 9-3 Smalltalk implementation of accurate statistical moments

```
Object subclass: #PMStatisticalMoments
   instanceVariableNames: 'moments'
   classVariableNames: ''
   package: 'Math-StatisticalMoments'
```

```
PMStatisticalMoments class >> new
    ^ self new: 4
```

```
PMStatisticalMoments class >> new: anInteger
    ^ super new initialize: anInteger + 1
```

```
PMStatisticalMoments >> accumulate: aNumber
    | correction n n1 oldSums pascal nTerm cTerm term |
    n := moments at: 1.
    n1 := n + 1.
    correction := ((moments at: 2) - aNumber) / n1.
    oldSums := moments copyFrom: 1 to: moments size.
    moments
        at: 1 put: n1;
        at: 2 put: (moments at: 2) - correction.
    pascal := Array new: moments size.
    pascal atAllPut: 0.
    pascal
        at: 1 put: 1;
        at: 2 put: 1.
    nTerm := -1.
    cTerm := correction.
    n1 := n / n1.
    n := n negated.
    3 to: moments size
        do:
            [:k |
            cTerm := cTerm * correction.
            nTerm := n * nTerm.
            term := cTerm * (1 + nTerm).
            k to: 3
                by: -1
                do:
                    [:l |
                    pascal at: l put: (pascal at: l - 1) + (pascal
                                                                at:
l).
                    term := (pascal at: l) * (oldSums at: l) + term.
                    oldSums at: l put: (oldSums at: l) * correction
].
            pascal at: 2 put: (pascal at: 1) + (pascal at: 2).
            moments at: k put: term * n1 ]
```

```
PMStatisticalMoments >> average
    self count = 0 ifTrue: [ ^nil ].
    ^ moments at: 2
```

```
PMStatisticalMoments >> centralMoment: anInteger
    ^ moments at: anInteger + 1
```

```
PMStatisticalMoments >> count
    ^ moments at: 1
```

```
PMStatisticalMoments >> errorOnAverage
    ^ (self variance / self count) sqrt
```

```
PMStatisticalMoments >> initialize: anInteger
    moments := Array new: anInteger.
    self reset.
    ^ self
```

```
PMStatisticalMoments >> kurtosis
    | n n1 n23 |
    n := self count.
    n < 4 ifTrue: [^nil].
    n23 := (n - 2) * (n - 3).
    n1 := n - 1.
    ^ ((moments at: 5) * n squared * (n + 1) / (self variance squared
                                                    * n1)
        - (n1 squared * 3)) / n23
```

```
PMStatisticalMoments >> reset
    moments atAllPut: 0
```

```
PMStatisticalMoments >> skewness
    | n v |
    n := self count.
    n < 3 ifTrue: [^nil].
    v := self variance.
    ^ (moments at: 4) * n squared / ((n - 1) * (n - 2) * (v sqrt *
    v))
```

```
PMStatisticalMoments >> standardDeviation
    ^ self variance sqrt
```

```
PMStatisticalMoments >> unnormalizedVariance
    ^ (self centralMoment: 2) * self count
```

```
PMStatisticalMoments >> variance
    | n |
    n := self count.
    n < 2
        ifTrue: [ ^nil].
    ^ self unnormalizedVariance / ( n - 1)
```

The class PMFixedStatisticalMoments is a specialization of the class PMStatisticalMoments for a fixed number of central moments going up to the 4 order.

The class creation method new: is barred from usage as the class can only be used for a fixed number of moment orders. As a consequence the default creation method must be redefined to delegate the parametric creation to the method of the superclass.

Listing 9-4 Smalltalk implementation of accurate statistical moments with fixed orders

```
PMStatisticalMoments subclass: #PMFixedStatisticalMoments
   instanceVariableNames: ''
   classVariableNames: ''
   package: 'Math-StatisticalMoments'
```

```
PMFixedStatisticalMoments class >> new
    ^ super new: 4
```

```
PMFixedStatisticalMoments class >> new: anInteger
    ^ self error: 'Illegal creation message for this class'
```

```
PMFixedStatisticalMoments >> accumulate: aNumber
    | correction n n1 c2 c3 |
    n := moments at: 1.
    n1 := n + 1.
    correction := ((moments at: 2) - aNumber) / n1.
    c2 := correction squared.
    c3 := c2 * correction.
    moments
        at: 5
            put: ((moments at: 5) + ((moments at: 4) * correction *
                                                                4)
                    + ((moments at: 3) * c2 * 6) + (c2 squared * (n
                                                 squared * n + 1)))
                    * n / n1;
        at: 4
            put: ((moments at: 4) + ((moments at: 3) * correction *
                                                               3)
                    + (c3 * (1 - n squared))) * n
                    / n1;
        at: 3 put: ((moments at: 3) + (c2 * (1 + n))) * n / n1;
        at: 2 put: (moments at: 2) - correction;
        at: 1 put: n1
```

9.3 Histograms

Whereas statistical moments provides a quick way of obtaining information about the distribution of a measured random variable, the information thus provided is rather terse and quite difficult to interpret by humans. Histograms provide a more complete way of analyzing an experimental distribution. A histogram has a big advantage over statistical moments: it can easily be represented graphically. Figure 9-5 shows a typical histogram.

A histogram is defined by three main parameters: x_{min}, the minimum of all values accumulated into the histogram, w, the bin width and n, the number of bins. A bin is defined as an interval. The i bin of a histogram is the interval $[x_{\text{min}} + (i-1)\,w, x_{\text{min}} + iw[$. The customary convention is that the lower limit is included in the interval and the higher limit excluded from the interval. The bin contents of a histogram — or histogram contents for short — is the number of times a value falls within each bin interval. Sometimes, a histogram is defined by the minimum and maximum of the accumulated values and the number of bins. The bin width is then computed as:

$$w = \frac{x_{\text{max}} - x_{\text{min}}}{n}, \tag{9.17}$$

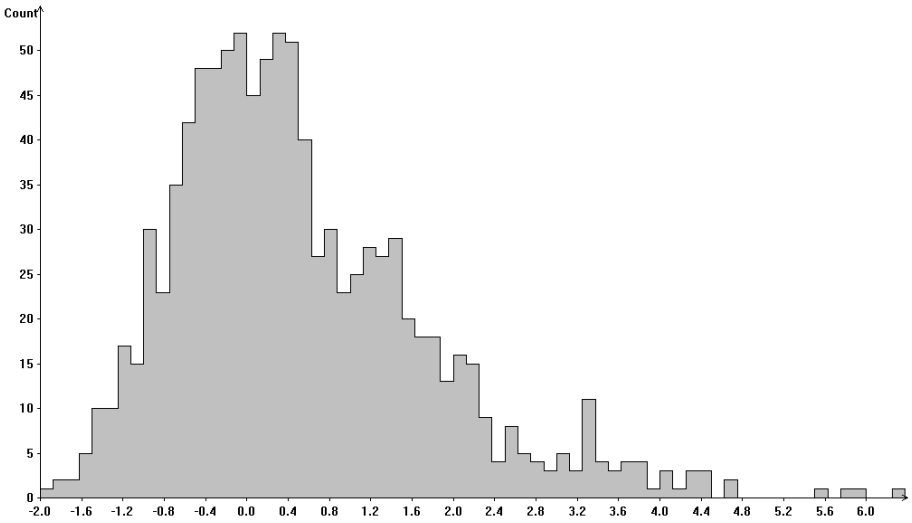


Figure 9-5 A typical histogram

where $x_{\max}$ is the maximum of the accumulated values.

In section 10.10 we shall need the error on the contents of a histogram. In absence of any systematic effects[4] the contents of each bin are distributed according to a Poisson distribution. The standard deviation of a Poisson distribution is the square root of the average. The standard deviation is used as an estimator of the error on the bin contents[5]. If n_i is the content of the i bin of the histogram, the estimated error on the contents is $\sqrt{n_i}$.

To obtain more information about the measured distribution, one can also keep track of the number of values falling outside of the histogram limits. The underflow of a histogram is defined as the number of values falling below the minimum of the accumulated values. Similarly, the overflow of a histogram is defined as the number of values falling on[6] or above the maximum of the accumulated values.

Histograms — implementation

Our implementation of histogram also accumulates the values into statistical moments. One can in principle compute the statistical moments of the mea-

[4] A good example of systematic effect is when values are computed from measurements made with an ADC. In this case, the integer rounding of the ADC may interfere with the bin sorting of the histogram.

[5] This is not a contradiction to what was said in section 9.1: the bin content is not an average, but a counting

[6] This is different from the definition of the underflow to be consistent with the fact that the definition of a bin interval is open ended at the upper limit.

sured distribution from the histogram contents. This determination, however, depends strongly on the bin width, especially if the bin width is large compared to the standard deviation. Thus, it is preferable to use the original data when accumulating the statistical moments. The net result is that a histogram has the same polymorphic behavior as a statistical moment.

When defining a histogram, the histogram limits — x_{min} and x_{max} — must be known in advance. This is not always practical since it implies a first scan of the measurements to determine the limits and a second scan to perform the accumulation into the defined histogram. Thus, our implementation offers the possibility of defining a histogram without predefined limits. In this mode, the first values are cached into an array until a sufficient number of data is available. When this happens, the histogram limits are determined from the data and the cached values are accumulated.

There are some cases when one would like to accumulates all the values within the histogram limits. The proposed implementation allows this by changing the histogram limits accordingly when a new value falls outside of the current histogram limits. When a histogram is accumulated in this mode the underflow and overflow counts are always zero.

When the histogram limits are computed automatically, it can happen that these limits have odd values. For example, if the minimum value is 2.13456 and the maximum value is 5.1245, selecting a number of bins of 50 would yield a bin width of 0.0597988. Of course such value for the bin width is quite undesirable in practice. A similar thing can happen if the application creating the histogram obtains the minimum and maximum values from a computation or an automatic measuring device. To avoid such silly parameters, our implementation computes a reasonable limit and bin width by rounding the bin width to the nearest reasonable scale at the order of magnitude[7] of the bin with. The possible scales are chosen to be easily computed by a human. In our example, the order of magnitude is -2. The bin width is then selected to be 0.075 and the minimum and maximum are adjusted to be integral multiples of the bin width enclosing the given limits. In our example, there are 2.1 and 5.175 and the number of bins becomes 41 instead of 50.

Listing **??** shows the implementation of a histogram in Smalltalk. The following code shows how to use the class PMHistogram to accumulate measurements into a histogram.

```
| histogram valueStream |
histogram := PMHistogram new.
[ valueStream atEnd ]
whileFalse: [ histogram accumulate: valueStream next ].
```

<*printing or display of the histogram*>

[7] Let us recall that the order of magnitude is the power of ten of a number.

This example assumes that the measurement of the random variable are obtained from a stream. The exact implementation of the stream is not shown here.

After the declarative statements, the first executable statement creates a new instance of the class PMHistogram with the default settings: automatic determination of the limits for 50 desired bins. The next two lines are the accumulation proper using a whileFalse: construct and the general behavior of a stream. This code is very similar to the code example presented in section 9.1. Extracting the parameters of the distribution can also be performed from the histogram.

The next example shows how to declare a histogram with given limits (2 and 7) and a desired number of bins of 50:

```
 | histogram valueStream |
 histogram := PMHistogram new.
 histogram setRangeFrom: 2.0 to: 7.0 bins: 100.
\hfil {\texttt<\textsl the rest is identical to the previous
    example\texttt
>}\hfil
```

The class PMHistogram has the following instance variables:

minimum the minimum of the accumulated values, that is $x_{\min}$,

binWidth the bin width, that is w,

overflow a counter to accumulate the overflow of the histogram,

underflow a counter to accumulate the underflow of the histogram,

moments an instance of the class PMFixedStatisticalMoments to accumulate statistical moments up to the 4 order (*c.f.* section 9.2) with minimal rounding errors.

contents the contents of the histogram, that is an array of integers,

freeExtent a Boolean flag denoting whether the limits of the histogram can be adjusted to include all possible values,

cacheSize the size of the cache allocated to collect values for an automatic determination of the histogram limits,

desiredNumberOfBins the number of bins desired by the calling application.

Since there are many ways to declare a histogram, there is a single creation method new, which calls in turn a single standard initialization method initialize. In this mode. the histogram is created with undefined limits — that is, the first accumulated values are cached until a sufficient number is available for an automatic determination of the limits — and a default number of bins. The default number of bins is defined by the class method defaultNumberOfBins.

Four methods allow to change the default initialization.

The method setRangeFrom:to:bins: allows the definition of the parameters x_{min}, x_{max} and n, respectively. The method setWidth:from:bins: allows the definition of the parameters w, x_{min} and n, respectively. In both cases, the histogram limits and number of bins are adjusted to reasonable values as explained at the end of section 9.3. These methods generate an error if the histogram is not cached, as limits cannot be redefined while the histogram is accumulating. The method setDesiredNumberOfBins: allows to overrule the default number of bins. Finally, the method freeExtent: takes a Boolean argument to define whether or not the limits must be adjusted when an accumulated value falls outside of the histogram limits. This method generates an error if any count has been accumulated in the underflow or overflow.

The method accumulate is used to accumulate the values. If the histogram is still cached — that is when values are directly accumulated into the cache for later determination of the limits — accumulation if delegated to the method accumulateInCache:. In this mode, the instance variable contents is an OrderedCollection collecting the values. When the size of the collection is reaching the maximum size allocated to the cache, limits are computed and the cache is flushed. In direct accumulation mode, the bin index corresponding to the value is computed. If the index is within the range, the value is accumulated. Otherwise it is treated like an overflow or an underflow. The method processOverflows:for: handles the case where the accumulated values falls outside of the histogram limits. If histogram limits cannot be adjusted it simply counts the overflow or the underflow. Otherwise processing is delegated to the methods growsContents:, growsPositiveContents and growsNegativeContents, which adjust the histogram limits according to the accumulated value.

The adjustment of the histogram limits to reasonable values is performed by the method adjustDimensionUpTo:. This is made when the limits are determined automatically. This method is also used when the limits are specified using one of the initialization methods.

There are many methods used to retrieve information from a histogram. Enumerating them here would be too tedious. Method names are explicit enough to get a rough idea of what each method is doing; looking at the code should suffice for a detailed understanding. The reader should just note that all methods retrieving the parameters of the distribution, as discussed in section 9.1, are implemented by delegating the method to the instance variable moments.

The iterator method pointsAndErrorsDo: is used for maximum likelihood fit of a probability distribution. Smalltalk implementation of maximum likelihood fit is discussed in section 10.10.

Listing 9-6 Smalltalk implementation of histograms

```
Object subclass: #PMHistogram
   instanceVariableNames: 'minimum binWidth overflow underflow
    moments contents freeExtent cacheSize desiredNumberOfBins'
   classVariableNames: ''
   package: 'Math-StatisticalMoments'
```

```
PMHistogram class >> defaultCacheSize
    ^ 100
```

```
PMHistogram class >> defaultNumberOfBins
^ 50
```

```
PMHistogram class >> integerScales
   ^ self
```

```
PMHistogram >> accumulate: aNumber
    | bin |
    self isCached
        ifTrue: [ ^self accumulateInCache: aNumber ].
    bin := self binIndex: aNumber.
    ( bin between: 1 and: contents size)
        ifTrue: [ contents at: bin put: ( contents at: bin) + 1.
                    moments accumulate: aNumber.
                ]
        ifFalse:[ self processOverflows: bin for: aNumber ].
```

```
PMHistogram >> accumulateInCache: aNumber
    contents add: aNumber.
    contents size > cacheSize
        ifTrue: [ self flushCache ].
```

```
PMHistogram >> adjustDimensionUpTo: aNumber
    | maximum |
    binWidth := self roundToScale: ( aNumber - minimum) /

    desiredNumberOfBins.
    minimum := ( minimum / binWidth) floor * binWidth.
    maximum := ( aNumber / binWidth) ceiling * binWidth.
    contents := Array new: ( ( maximum - minimum) / binWidth)

    ceiling.
    contents atAllPut: 0.
```

```
PMHistogram >> average
    ^ moments average
```

```
PMHistogram >> binIndex: aNumber
    ^( ( aNumber - minimum) / binWidth) floor + 1
```

```
PMHistogram >> binWidth
    self isCached
        ifTrue: [ self flushCache].
    ^binWidth
```

```
PMHistogram>>collectIntegralPoints: aBlock
    | answer bin lastContents integral norm x |
    self isCached
        ifTrue: [ self flushCache].
    answer := OrderedCollection new: ( contents size * 2 + 1).
    bin := self minimum.
    answer add: ( aBlock value: bin @ 0).
    integral := self underflow.
    norm := self totalCount.
    contents do:
        [ :each |
          integral := integral + each.
          x := integral / norm.
          answer add: ( aBlock value: bin @ x).
          bin := bin + binWidth.
          answer add: ( aBlock value: bin @ x).
        ].
    answer add: ( aBlock value: bin @ 0).
    ^ answer asArray
```

```
PMHistogram>>collectPoints: aBlock
    | answer bin lastContents |
    self isCached
        ifTrue: [ self flushCache ].
    answer := OrderedCollection new: ( contents size * 2 + 1).
    bin := self minimum.
    answer add: (aBlock value: bin @ 0).
    contents do:
        [ :each |
          answer add: (aBlock value: bin @ each).
          bin := bin + binWidth.
          answer add: (aBlock value: bin @ each).
        ].
    answer add: (aBlock value: bin @ 0).
    ^ answer asArray
```

```
PMHistogram>>count
    ^ moments count
```

```
PMHistogram>>countAt: aNumber
    | n |
    n := self binIndex: aNumber.
    ^ (n between: 1 and: contents size)
            ifTrue: [ contents at: n ]
            ifFalse: [ 0 ]
```

```
PMHistogram>>countOverflows: anInteger
    anInteger > 0
        ifTrue: [ overflow := overflow + 1 ]
        ifFalse:[ underflow := underflow + 1 ].
```

```
PMHistogram >> countsBetween: aNumber1 and: aNumber2
    | n1 n2 answer |
    n1 := self binIndex: aNumber1.
    n2 := self binIndex: aNumber2.
    answer := ( contents at: n1) * ( ( binWidth * n1 + minimum) -
                                                    aNumber1) /
    binWidth.
    n2 > contents size
        ifTrue: [ n2 := contents size + 1]
        ifFalse:[ answer := answer + ( ( contents at: n2) * (
      aNumber2 - ( binWidth * ( n2 - 1) + self maximum)) /
    binWidth)].
    ( n1 + 1) to: ( n2 - 1) do: [ :n | answer := answer + ( contents
                                                            at:
    n)].
    ^ answer
```

```
PMHistogram >> countsUpTo: aNumber
    | n answer |
    n := self binIndex: aNumber.
    n > contents size
        ifTrue: [ ^self count].
    answer := ( contents at: n) * ( aNumber - ( binWidth * ( n - 1) +
                                            self minimum)) /
    binWidth.
    1 to: ( n - 1) do: [ :m | answer := answer + ( contents at: m)].
    ^ answer + underflow
```

```
PMHistogram >> errorOnAverage
    ^ moments errorOnAverage
```

```
PMHistogram >> flushCache
    | maximum values |
    minimum isNil
        ifTrue: [ minimum := contents isEmpty ifTrue: [ 0]

                                            ifFalse:[ contents
    first].
                    ].
    maximum := minimum.
    contents do:
        [ :each |
          each < minimum
            ifTrue: [ minimum := each]
            ifFalse:[ each > maximum
                            ifTrue: [ maximum := each].
                    ].
        ].
    maximum = minimum
        ifTrue: [ maximum := minimum + desiredNumberOfBins].
    values := contents.
```

```
    self adjustDimensionUpTo: maximum.
    values do: [ :each | self accumulate: each ].
```

```
PMHistogram>>freeExtent: aBoolean
    ( underflow = 0 and: [ overflow = 0])
        ifFalse: [ self error: 'Histogram extent cannot be

    redefined'].
    freeExtent := aBoolean.
```

```
PMHistogram>>growContents: anInteger
    anInteger > 0
        ifTrue: [ self growPositiveContents: anInteger ]
        ifFalse: [ self growNegativeContents: anInteger ].
```

```
PMHistogram>>growNegativeContents: anInteger
    | n newSize newContents |
    n := 1 - anInteger.
    newSize := contents size + n.
    newContents := Array new: newSize.
    newContents
            at: 1 put: 1;
            replaceFrom: 2 to: n withObject: 0;
            replaceFrom: ( n + 1) to: newSize with: contents.
    contents := newContents.
    minimum := ( anInteger - 1) * binWidth + minimum.
```

```
PMHistogram>>growPositiveContents: anInteger
    | n newContents |
    n := contents size.
    newContents := Array new: anInteger.
    newContents
            replaceFrom: 1 to: n with: contents;
            replaceFrom: ( n + 1) to: ( anInteger - 1) withObject: 0;
            at: anInteger put: 1.
    contents := newContents.
```

```
PMHistogram>>initialize
    freeExtent := false.
    cacheSize := self class defaultCacheSize.
    desiredNumberOfBins := self class defaultNumberOfBins.
    contents := OrderedCollection new: cacheSize.
    moments := DhbFixedStatisticalMoments new.
    overflow := 0.
    underflow := 0
```

```
PMHistogram>>inverseDistributionValue: aNumber
    | count x integral |
    count := self count * aNumber.
    x := self minimum.
    integral := 0.
    contents do:
```

```
        [ :each | | delta |
          delta := count - integral.
          each > delta
            ifTrue: [ ^self binWidth * delta / each + x].
          integral := integral + each.
          x := self binWidth + x.
        ].
    ^ self maximum
```

```
PMHistogram>>isCached
    ^ binWidth isNil
```

```
PMHistogram>>isEmpty
    ^ false
```

```
PMHistogram>>kurtosis
    ^ moments kurtosis
```

```
PMHistogram>>lowBinLimitAt: anIntege
    ^ (anInteger - 1) * binWidth + minimum
```

```
PMHistogram>>maximum
    self isCached
        ifTrue: [ self flushCache ].
    ^ contents size * binWidth + minimum
```

```
PMHistogram>>maximumCount
    self isCached
        ifTrue: [ self flushCache ].
    ^contents inject: ( contents isEmpty ifTrue: [ 1] ifFalse:[
                                                    contents at:
    1])
                    into: [ :max :each | max max: each]
```

```
PMHistogram>>minimum
    self isCached
        ifTrue: [ self flushCache ].
    ^ minimum
```

```
PMHistogram>>overflow
    ^ overflow
```

```
PMHistogram>>processOverflows: anInteger for: aNumber
    freeExtent
        ifTrue: [ self growContents: anInteger.
                    moments accumulate: aNumber
                ]
        ifFalse:[ self countOverflows: anInteger ].

\begin{displaycode}{Smalltalk}
PMHistogram>>setDesiredNumberOfBins: anInteger
    anInteger > 0
        ifFalse:[ self error: 'Desired number of bins must be
```

```
    positive'].
    desiredNumberOfBins := anInteger.
```

```
PMHistogram>>setRangeFrom: aNumber1 to: aNumber2 bins: anInteger
    self isCached
        ifFalse: [ self error: 'Histogram limits cannot be

    redefined'].
    minimum := aNumber1.
    self setDesiredNumberOfBins: anInteger;
           adjustDimensionUpTo: aNumber2.
```

```
PMHistogram>>setWidth: aNumber1 from: aNumber2 bins: anInteger
    self isCached
        ifFalse: [ self error: 'Histogram limits cannot be

    redefined'].
    minimum := aNumber2.
    self setDesiredNumberOfBins: anInteger;
           adjustDimensionUpTo: ( aNumber1 * anInteger + aNumber2).
```

```
PMHistogram>>skewness
    ^ moments skewness
```

```
PMHistogram>>standardDeviation
    ^ moments standardDeviation
```

```
PMHistogram>>totalCount
    ^ moments count + underflow + overflow
```

```
PMHistogram>>underflow
    ^ underflow
```

```
PMHistogram>>variance
    ^ moments variance
```

9.4 Random number generator

When studying statistical processes on a computer one often has to simulate the behavior of a random variable[8]. As we shall see in section 9.5 it suffice to implement a random generator for a uniform distribution, that is a random variable whose probability density function is constant over a given interval. Once such an implementation is available, any probability distribution can be simulated.

Linear congruential random generators

The most widely used random number generators are linear congruential random generators. Random numbers are obtained from the following series

[8] Another wide use for random number generators are games.

[?]:

$$X_{n+1} = (aX_n + c) \bmod m, \tag{9.18}$$

where m is called the modulus, a the multiplier and c the increment. By definition, we have $0 \leq X_n < m$ for all n. The numbers X_n are actually pseudo-random numbers since, for a given modulus, multiplier and increment, the sequence of numbers $X_1, \ldots, X_n$ is fully determined by the value X_0. The value X_0 is called the seed of the series. In spite of its reproducibility the generated series behaves very close to that of random variable uniformly distributed between 0 and $m - 1$. Then the following variable:

$$x_n = \frac{X_n}{m}, \tag{9.19}$$

is a random rational number uniformly distributed between 0 and 1, 1 excluded.

In practice, the modulus, multiplier and increment must be chosen quite carefully to achieve a good randomness of the series Don Knuth [?] gives a series of criteria for choosing the parameters of the random number generator. If the parameters are correctly chosen, the seed X_0 can be assigned to any value.

Additive sequence generators

Another class of random generators are additive sequence generators [?]. The series of pseudo-random numbers is generated as follows:

$$X_n = (X_{n-l} + X_{n-k}) \bmod m, \tag{9.20}$$

where m is the modulus as before and l and k are two indices such that $l < k$. These indices must be selected carefully. [?] contains a table of suitable indices. The initial series of numbers $X_1, \ldots, X_k$ can be any integers not all even.

Generators based on additive sequences are ideal to generate floating point numbers. If this case, the modulo operation on the modulus is not needed. Instead, one simply checks whether or not the newly generated number is larger than 1. Thus, the series becomes:

$$\begin{aligned} y_n &= x_{n-l} + x_{n-k}, \\ x_n &= \begin{cases} y_n & \text{if } y_n < 1, \\ y_n - 1 & \text{if } y_n \geq 1, \end{cases} \end{aligned} \tag{9.21}$$

It is clear that the evaluation above is much faster than that of equation 9.18. In practice, the additive sequence generator is about 4 times faster. In addition, the length of the sequence is larger than that of a congruential random generator with the same modulus.

In our implementation we have selected the pair of numbers $(24, 55)$ corresponding to the generator initially discovered by G.J. Mitchell and D.P. Moore[**?**]. The corresponding sequence has a length of $2^{55} - 1$. In our tests (*c.f.* below) we have found that the randomness of this generator is at least as good as that of the congruential random generator. The initial series $x_1, \ldots, x_{55}$ is obtained from the congruential random generator.

In [**?**] Don Knuth describes a wealth of test to investigate the randomness of random number generators. Some of these tests are also discussed in [**?**]. To study the goodness of our proposed random generators, we have performed two types of tests: a χ^2 test and a correlation test.

The χ^2 test is performed on a histogram, in which values generated according to a probability distributions have been accumulated. Then, a χ^2 confidence level (*c.f.* section 10.3) is calculated against the theoretical curve computed using the histogram bin width, the number of generated values and the parameters of the distribution. A confidence level should be larger than 60% indicates that the probability distribution is correctly generated. When using distributions requiring the generation of several random numbers to obtain one value — Normal distribution (2 values), gamma distribution (2 values) and beta distribution (4 values) — one can get a good confidence that short term correlations[9] are not creating problems. The code for this test is given in the code examples **??**. In this test the Mitchell-Moore generator gives results similar to that of the congruential random generator.

The correlation test is performed by computing the covariance matrix (*c.f.* section 12.2) of vectors of given dimension (between 5 and 10). The covariance matrix should be a diagonal matrix with all diagonal elements equal to $1/12$, the variance of a uniform distribution. Deviation from this theoretical case should be small. Here longer correlations can be investigated by varying the dimension of the generated vectors. In this test too, the Mitchell-Moore generator gives results similar to that of the congruential random generator.

Bit-pattern generators

The generators described above are suitable to the generation of random values, but not for the generation of random bit patterns [**?**], [**?**]. The generation of random bit patterns can be achieved with generator polynomials. Such polynomials are used in error correction codes for their abilities to produce sequences of numbers with a maximum number of different bits. For example the following polynomial

$$G(x) = x^{16} + x^{12} + x^5 + 1, \tag{9.22}$$

[9] Pseudo random number generators have a tendency to exhibit correlations in the series. That is, the number X_n can be correlated to the number X_{n-k} for each n and a given k.

is a good generator for random patterns of 16 bits[10]. Of course, the evaluation of equation 9.22 does not require the computation of the polynomial. The following algorithm can be used:

1. Set X_{n+1} to X_n shifted by one position to the left and truncated to 16 bits ($X_{n+1} = 2X_n \bmod 2^{16}$),
2. If bit 15 (least significant bit being 0) of X_n is set, set X_{n+1} to the bit wise exclusive OR of X_{n+1} with `0x1021`.

Other polynomials are given in [**?**].

Random bit patterns are usually used to simulate hardware behavior. They are rarely used in statistical analysis. A concrete implementation of a random bit pattern generator is left as an exercise to the reader.

Random number generator — Smalltalk implementation

Figure 9-1 with the boxes **C**ongruenti and **M**itchellMo grayed.

Listing 9-7 shows the implementation of a congruential random generator in Smalltalk. Listing **??** shows the implementation of a additive sequence random generator in Smalltalk. Listing 9.4 shows usage of the generator for standard use.

The class PMCongruentialRandomNumberGenerator has three public methods:

`value` returns the next random number of the series, that is X_n, a number between 0 and m,

`floatValue` returns the next random floating number, that is the value X_n/m,

`integerValue:` returns a random integer, whose values are between 0 and the specified argument.

When calling any of the above methods, the next random number of the series is obtained using equation 9.18.

There are several ways of using a random number generator. If there is no specific requirement the easiest approach is to use the instance provided by default creation method (new) returning a singleton. The next example shows how to proceed assuming the application uses the values generated by the random generator directly:

```
 | generator x |
 generator := PMCongruentialRandomNumberGenerator new.
\hfil{\texttt <\textsl Here is where the generator is used\texttt
>}\hfil

 x := generator value.
```

[10] *c.f.* O. Yamada, K. Yamazaki and D.H.Besset, *An Error-Correction Scheme for an Optical Memory Card System*, 1990 International Symposium on Information Theory and its Applications (ISITA'90), Hawaii, USA, November 27-30, 1990.

If one needs several series which must be separately reproducible, one can assign several generators, one for each series. The application can use predefined numbers for the seed of each series. Here is a complete example assuming that the generated numbers must be floating numbers.

```
 | generators seeds index x |

 {\texttt seeds := <\textsl an array containing the desired
    seeds\texttt >}

 generators := seeds collect:
           [ :each | PMCongruentialRandomNumberGenerator seed: each].

\noindent\hskip 5ex{\texttt <\textsl Here is where the various
    generators
are used\texttt
>}\hfil \\
\hskip 5ex{\texttt <index \textsl is the index of the desired
    series\texttt
>}\hfil

 x := ( generators at: index) floatValue.
```

In game applications, it is of course not desirable to have a reproducible series. In this case, the easiest way is to use the time in milliseconds as the seed of the series. This initial value is sufficiently hard to reproduce to give the illusion of randomness. Furthermore the randomness of the series guarantees that two series generated at almost the same time are quite different. Here is how to do it.

```
 | generator x |
 generator := PMCongruentialRandomNumberGenerator
                        seed: Time millisecondClockValue.

\hfil{\texttt <\textsl Here is where the generator is used\texttt
>}\hfil

 x := (generator integerValue: 20) + 1.
```

In this last example, the generated numbers are integers between 1 and 20.

Implementation

The class PMCongruentialRandomNumberGenerator has the following instance variables:

constant the increment c,

modulus the modulus m,

multiplicator the multiplier a and

seed the last generated number X_{n-1}.

There are three instance creation methods. The method new returns a Singleton instance containing parameters from [**?**]. The method seed: allows one to create an instance to generate a series of random number starting at a given seed. Finally the method constant:multiplicator:modulus: creates a congruential random number generator based on supplied parameters. Readers tempted to use this method are strongly advised to read [**?**] and the references therein thoroughly. Then, they should perform tests to verify that their parameters are indeed producing a series with acceptable randomness.

The modulus of the standard parameters has 32 bits. In the Smalltalk implementation, however, the evaluation of equation 9.18 generates integers larger than 32 bits. As a result, the generation of the random numbers is somewhat slow, as it is using multiple precision integers. Using floating number[11] does not disturb the evaluation of equation 9.18 and is significantly faster since floating point evaluation is performed on the hardware. The generation of random numbers using floating point parameters is about 3 times faster than with integer parameters. This can easily be verified by the reader.

Listing 9-7 Smalltalk implementation of congruential random number generators

```
Object subclass: #PMCongruentialRandomNumberGenerator
    instanceVariableNames: 'constant modulus multiplicator seed'
    classVariableNames: 'UniqueInstance'
    package: #'Math-Distributions'
```

```
PMCongruentialRandomNumberGenerator class >> constant: aNumber1
    multiplicator: aNumber2 modulus: aNumber3
    ^ super new
        initialize: aNumber1
        multiplicator: aNumber2
        modulus: aNumber3
```

```
PMCongruentialRandomNumberGenerator class >> new
    UniqueInstance isNil
        ifTrue: [ UniqueInstance := super new initialize.
                    UniqueInstance setSeed: 1 ].
    ^ UniqueInstance
```

```
PMCongruentialRandomNumberGenerator class >> seed: aNumber
    ^ super new initialize; setSeed: aNumber; yourself
```

```
PMCongruentialRandomNumberGenerator >> floatValue
    ^ self value asFloat / modulus
```

```
PMCongruentialRandomNumberGenerator >> initialize
    self initialize: 2718281829.0 multiplicator: 3141592653.0
                                                modulus: 4294967296.0
```

[11] The author is grateful to Dave N. Smith of IBM for this useful tip.

```
PMCongruentialRandomNumberGenerator >> initialize: aNumber1
    multiplicator: aNumber2 modulus: aNumber3
    constant := aNumber1.
    modulus := aNumber2.
    multiplicator := aNumber3.
    self setSeed: 1.
```

```
PMCongruentialRandomNumberGenerator >> integerValue: anInteger
    ^ (self value \\ ( anInteger * 1000)) // 1000
```

```
PMCongruentialRandomNumberGenerator >> setSeed: aNumber
    seed := aNumber
```

```
PMCongruentialRandomNumberGenerator >> value
    seed := (seed * multiplicator + constant) \\ modulus.
    ^ seed
```

The class PMMitchellMooreGenerator implements a random number generator with additive sequence. It has two public methods:

floatValue returns the next random floating number, that is the value x_n,

integerValue: returns a random integer, whose values are between 0 and the specified argument.

When calling any of the above methods, the next random number of the series is obtained using equation 9.21. The series of generated numbers are all floating points.

The creation methods new and seed: are used exactly as the corresponding methods of the class PMCongruentialRandomNumberGenerator. Please refer to the code examples **??** and **??**. Both methods use the congruential random number generator to generate the initial series of numbers $x_1, \ldots, x_{55}$. The class method constants:lowIndex: offers a way to define the numbers k and l as well as the initial series of numbers. The reader wishing to use this method should consult the table of *g*ood indices k and l in [**?**].

Listing 9-8 m

```
alltalk implementation of an additive sequence random number
    generator
\label{ls:randomseq}
%\input{Smalltalk/Statistics/DhbMitchellMooreGenerator}
```

For simple simulation, one wishes to generate a random number — floating or integer — within certain limits. Here are convenience methods implemented for the class Number and Integer. These methods frees the user from keeping track of the instance of the random number generator. For example, the following Smalltalk expression

```
50 random
```

generates an integer random number between 0 and 49 included. Similarly the following Smalltalk expression

```
2.45 random
```

generates a floating random number between 0 and 2.45 excluded. Finally the following Smalltalk expression

```
Number random
```

generates a floating random number between 0 and 1 excluded.

Listing 9-9 Smalltalk implementation of random number generators

```
Integer >> random
    ^PMMitchellMooreGenerator new integerValue: self
```

```
Number class >> random
    ^ PMMitchellMooreGenerator new floatValue
```

```
Number >> random
    ^ self class random * self
```

9.5 Probability distributions

A probability density function defines the probability of finding a continuous random variable within an infinitesimal interval. More precisely, the probability density function $P(x)$ gives the probability for a random variable to take a value lying in the interval $[x, x + dx[$. A probability density function is a special case of a one variable function described in section 2.1.

The moment of k^{th} order for a probability density function $P(x)$ is defined by:

$$M_k = \int x^k P(x)\, dx, \tag{9.23}$$

where the range of the integral is taken over the range where the function $P(x)$ is defined. By definition probability density functions are normalized, that is, M_0 is equal to 1.

As for statistical moments, one defines the mean or average of the distribution as:

$$\mu = M_1 = \int x P(x)\, dx. \tag{9.24}$$

Then the central moment of k^{th} order is defined by:

$$m_k = \int (x - \mu)^k P(x)\, dx. \tag{9.25}$$

In particular the variance is defined as m_2, the central moment of second order. The standard deviation, σ, is the square root of m_2. The skewness and

kurtosis[12] of a probability density function are respectively defined as:

$$\omega = \frac{m_3}{\sqrt[3/2]{m_2}} = \frac{m_3}{\sigma^3} \quad \text{and} \tag{9.26}$$

$$\kappa = \frac{m_4}{m_2^2} - 3 = \frac{m_4}{\sigma^4} - 3. \tag{9.27}$$

The distribution function, also called acceptance function or repartition function, is defined as the probability for the random variable to have a value smaller or equal to a given value. For a probability density function defined over all real numbers we have:

$$F(t) = \text{Prob}(x < t) = \int_{-\infty}^{t} P(x)\, dx. \tag{9.28}$$

If the probability density function $P(x)$ is defined over a restricted range, the lower limit of the integral in equation 9.28 must be changed accordingly. For example, if the probability density function is defined for $x \geq x_{\text{min}}$, the distribution function is given by:

$$F(t) = \text{Prob}(x < t) = \int_{x_{\text{min}}}^{t} P(x)\, dx. \tag{9.29}$$

Instead of the distribution function, the name centile is sometimes used to refer to the value of the distribution function expressed in percent. This kind of terminology is frequently used in medical and natural science publications. For example, a value x is said to be at the 10^{th} centile if $F(t) = 1/10$; in other words, there is a ten-percent chance of observing a value less than or equal to t[13].

The interval acceptance function measures the probability for the random variable to have a value between two given values. That is

$$F(x_1, x_2) = \text{Prob}(x_1 \leq x < x_2) = \int_{x_1}^{x_2} P(x)\, dx, \tag{9.30}$$

$$F(x_1, x_2) = F(x_2) - F(x_1). \tag{9.31}$$

If the integral of equation 9.28 can be resolved into a closed expression or if it has a numerical approximation, the evaluation of the interval acceptance function is made using equation 9.31. Otherwise the interval acceptance function must be evaluated using Romberg integration (*c.f.* section 6.4) applied to equation 9.30.

The inverse of the repartition function is frequently used. For example, in order to determine an acceptance threshold in a decision process, one needs to

[12] In old references the kurtosis, as defined here, is called excess; then, the kurtosis is defined as the square of the excess; [**?**] *e.g.*.

[13] A centile can also be quoted for a value relative to a set of observed values.

determine the variable t such that the repartition function is equal to a given value p. In other words, $t = F^{-1}(p)$. For example, the threshold of a coin detection mechanism to only reject 99.9% of the good coins is $F^{-1}(0.999)$. If the distribution function is not invertible, one can solve this equation using the Newton's zero-finding method exposed in section 5.3. Newton's zero-finding method is especially handy since the derivative of the function is known: it is the probability density function. Since $F(x)$ is strictly monotonous between 0 and 1 a unique solution is guaranteed for any p within the open interval $]0, 1[$. The initial value for the search can be obtained from Markov's inequality **[?]**, which can be written in the form:

$$t \leq \frac{\mu}{1 - F(t)} \tag{9.32}$$

If no closed expression exists for the distribution function (it is determined using numerical integration *e.g.*) the computation of the inverse value is best obtained by interpolating the inverse function over a set of by tabulated values (*c.f.* section 3).

The inverse of the distribution function is also used to generate a random variable distributed according to the distribution. Namely, if r is a random variable uniformly distributed between 0 and 1, then the variable $x = F^{-1}(r)$ is a random variable distributed according to the distribution whose distribution function is $F(x)$. In practice this method can only be used if a closed expression exists for the distribution function, otherwise the function must be tabulated and Newton interpolation can be used on the inverse function (*c.f.* section 3.3). For most known distributions, however, special algorithms exist to generate random values distributed according to a given distribution. Such algorithms are described in **[?]**. They are not discussed here, but the code is included in each implementation of the specific probability distribution.

In the rest of this chapter we shall present the distributions used in this book. Other important distributions are presented in appendix **??**.

Probability distributions — Smalltalk implementation

Figure 9-1 with the boxes **P**robability[and **P**robability[grayed.

Table 9-10 shows the description of the public methods of the implementation.

Depending on the distribution, closed expressions for the variance or the standard deviation exist. Here general methods are supplied to compute one from the other. Subclasses must implement at least one of them; otherwise a stack overflow will result.

Methods to compute skewness and kurtosis are supplied, but return nil in Smalltalk. A very general implementation could have used explicit integration. The integration interval, however, maybe infinite and a general integration strategy cannot easily be supplied. Subclasses are expected to implement both methods.

Table 9-10 Public methods for probability density functions

Description	Smalltalk
$P(x)$	value:
$F(x)$	distributionValue:
$F(x_1, x_2)$	acceptanceBetween:and:
$F^{-1}(x)$	inverseDistributionValue:
$x^{\dagger}$	random
$\bar{x}$	average
σ^2	variance
σ	standardDeviation
skewness	skewness
kurtosis	kurtosis

† x represents the random variable itself. In other words, the method random returns a random value distributed according to the distribution.

As we have quoted earlier a probability density function is a function, as defined in section 2.1. Since the distribution function is also a function, a Adapter must be provided to create a function (as defined in section 2.1) for the distribution function.

Listing **??** shows the implementation of a general probability density distribution in Smalltalk. The class PMProbabilityDensity is an abstract implementation. Concrete implementation of probability density distributions are subclass of it.

The method distributionValue: returning the value of the distribution function must be implemented by the subclass. The method to compute the interval acceptance functions is using equation 9.31 .

The inverse acceptance function is defined with two methods, one public and one private. The public method verifies the range of the argument, which must lie between 0 and 1. The private method uses the class PMNewtonZeroFinder discussed in section 5.3. The derivative needed by the Newton zero finder is the probability density function itself since, by definition, it is the derivative of the acceptance function (*c.f.* equation 9.28).

Finally the class creation method fromHistogram: creates a new instance of a probability distribution with parameters derived using a quick approximation from the data accumulated into the supplied histogram; the derivation assumes that the accumulated data are distributed according to the distribution. This method is used to compute suitable starting values for least square or maximum likelihood fits (*c.f.* chapter 10). The convention is that this methods returns nil if the parameters cannot be obtained. Thus, returning nil is the default behavior for the superclass since this method is specific to each distribution. The estimation of the parameters is usually made using the statistical moments of the histogram and comparing them to the analytical expression of the distribution's parameter.

Listing 9-11 m

```
alltalk implementation of a probability distribution
\label{ls:probdistr}
%\input{Smalltalk/Statistics/DhbProbabilityDensity}
```

The class PMProbabilityDensityWithUnknownDistribution is the abstract class for probability distribution having neither an analytical expression nor a numerical approximation for the distribution function.

Therefore, methods computing the acceptance function (distributionValue:) and interval acceptance (acceptanceBetween:and:) are using equations 9.28 and 9.30 respectively, using the class PMRombergIntegrator discussed in section 6.4. The lower limit of the integral for the distribution function — $x_{\min}$ of equation 9.29 — is defined by the method `lowestValue`. Since the majority of the probability density distributions are defined for non-negative numbers, this method returns 0. If the supplied default is not appropriate, the method `lowestValue` must be redefined by the subclass.

Listing 9-12 m

```
alltalk implementation of a probability distribution with unknown
distribution function\label{ls:probunkdistr}
%\input{Smalltalk/Statistics/DhbProbabilityDensityWithUnknownDistribution}
```

Listing **??** shows the implementation of the Adapter for the distribution function. The class PMProbabilityDistributionFunction has a single instance variable containing the corresponding probability density function. The creation method density: takes an instance of class PMProbabilityDensity as argument.

Listing 9-13 m

```
alltalk implementation of a probability distribution function
\label{ls:probdistrfun}
%\input{Smalltalk/Statistics/DhbProbabilityDistributionFunction}
```

9.6 Normal distribution

The normal distribution is the most important probability distribution. Most other distributions tend toward it when some of their parameters become large. Experimental data subjected only[14] to measurement fluctuation usually follow a normal distribution.

Table 9-14 shows the properties of the normal distribution. Figure 9-15 shows the well-known bell shape of the normal distribution for various values of the parameters. The reader can see that the peak of the distribution is always located at μ and that the width of the bell curve is proportional to σ.

[14] The presence of systematic errors is a notable exception to this rule.

Table 9-14 Properties of the Normal distribution

Range of random variable	$]-\infty, +\infty[$
Probability density function	$P(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$
Parameters	$-\infty < \mu < +\infty$ $0 < \sigma < +\infty$
Distribution function	$F(x) = \mathrm{erf}\left(\frac{x-\mu}{\sigma}\right)$ (*c.f.* section 2.4)
Average	μ
Variance	σ^2
Skewness	0
Kurtosis	0

Figure 9-1 with the box NormalDistribution grayed.

Normal distribution — Smalltalk implementation

Listing **??** shows the implementation of the normal distribution in Smalltalk.

The distribution function of the normal distribution can be computed with the error function (*c.f.* section 2.4). Therefore the class PMNormalDistribution is implemented as a subclass of PMProbabilityDensity.

Listing 9-16 m

```
alltalk implementation of the normal distribution \label{ls:normdist}
\input{Smalltalk/Statistics/DhbNormalDistribution}
```

9.7 Gamma distribution

The gamma distribution is used to describe the time between some task, for example the time between repairs.

The generalization of the gamma distribution to a range of the random variable of the type $[x_{\min}, +\infty[$ is called a Pearson type III distribution. The average of a Pearson type III distribution is $x_{\min} + \alpha\beta$. The central moments are the same as those of the gamma distribution.

Table 9-17 shows the properties of the gamma distribution. Figure 9-18 shows the shape of the gamma distribution for several values of the parameter α with $\beta = 1$. The shape of the distribution for values of the parameter β can be obtained by modifying the scale of the x-axis since β is just a scale factor of the random variable.

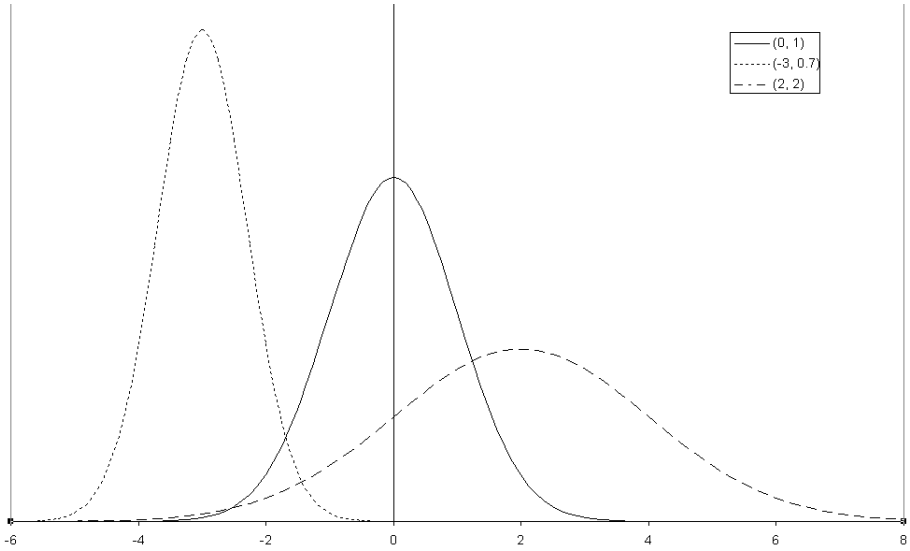


Figure 9-15 Normal distribution for various values of the parameters

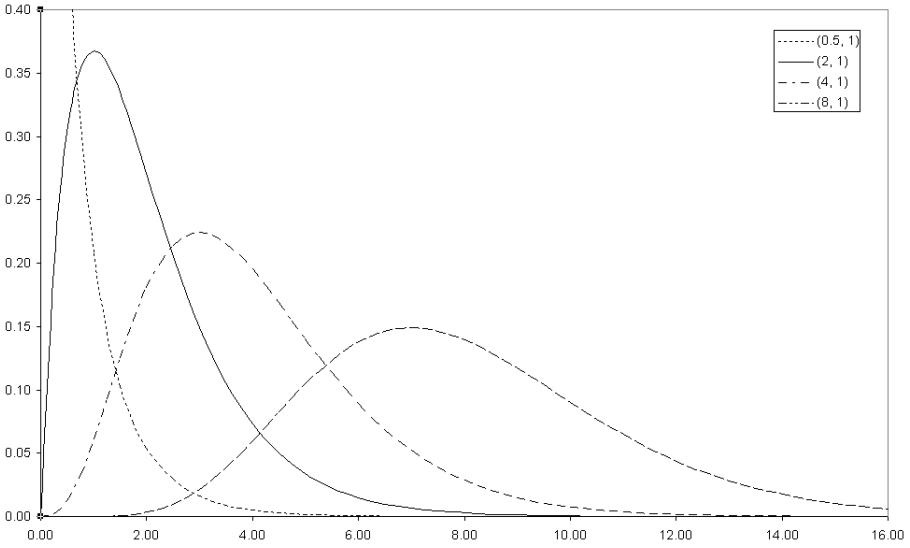


Figure 9-18 Gamma distribution for various values of α

Table 9-17 Properties of the gamma distribution

Range of random variable	$[0, +\infty[$
Probability density function	$P(x) = \frac{x^{\alpha-1}}{\beta^{\alpha}\Gamma(\alpha)} e^{-\frac{x}{\beta}}$
Parameters	$0 < \alpha < +\infty$ $0 < \beta < +\infty$
Distribution function	$F(x) = \Gamma\left(\frac{x}{\beta}, \alpha\right)$ (*c.f.* section 7.4)
Average	$\alpha\beta$
Variance	$\alpha\beta^2$
Skewness	$\frac{2}{\sqrt{\alpha}}$
Kurtosis	$\frac{6}{\alpha}$

Figure 9-1 with the box GammaDistribution grayed.

Gamma distribution — Smalltalk implementation

Listing **??** shows the implementation of the gamma distribution in Smalltalk.

The distribution function of the gamma distribution can be computed with the incomplete gamma function (*c.f.* section 2.5). Therefore the class PMGammaDistribution is implemented as a subclass of PMProbabilityDensity.

Listing 9-19 m

```
alltalk implementation of the gamma distribution \label{ls:gammadist}
\input{Smalltalk/Statistics/DhbGammaDistribution}
```

9.8 Experimental distribution

A histogram described in section 9.3 can be used as a probability distribution. After all, a histogram can be considered as the representation of a distribution, which has been measured experimentally.

If N is the total count of the histogram and n_i the count in bin number i the probability of finding a measurement within the bin number i is simply given by:

$$P_i = \frac{n_i}{N}. \tag{9.33}$$

If w is the width of each bin, the probability density function of the distribu-

tion measured by the histogram can be estimated by:

$$P(x) = \frac{P_i}{w} = \frac{n_i}{wN} \quad \text{where } i = \left\lfloor \frac{x - x_{\min}}{w} \right\rfloor. \tag{9.34}$$

Equation 9.34 is only valid for $x_{\min} \leq x < x_{\max}$. Outside of the histogram's limits there is no information concerning the shape of the probability density function.

The distribution function is computed by evaluating the sum of all bins located below the argument and by adding a correction computed by linear interpolation over the bin in which the value is located. Thus, we have:

$$F(x) = \frac{1}{N}\left(\sum_{j=1}^{i-1} n_j + \frac{x - x_i}{w} n_i\right) \quad \text{where } i = \left\lfloor \frac{x - x_{\min}}{w} \right\rfloor. \tag{9.35}$$

If $x < x_{\min}$, $F(x) = 0$ and if $x \geq x_{\max}$, $F(x) = 1$. A similar equation can be derived for the acceptance interval function.

Experimental distribution — General implementation

Figure 9-1 with the box **H**istogramm grayed.

Adding the responsibility of behaving like a probability density function to a histogram is not desirable. In a good object oriented design, objects should have only one responsibility or type of behavior.

Thus, a good object oriented implementation implements an Adapter pattern. One creates an object, having the behavior of a probability density function. A single instance variable inside this object refers to the histogram, over which the experimental distribution is defined. The Adapter object is a subclass of the abstract class describing all probability density functions.

The parameters of the distribution — average, variance, skewness and kurtosis — are obtained from the histogram itself.

The computation of the distribution and the interval acceptance function is delegated to the histogram. The floor operation of equation 9.35 is evaluated by the method binIndex of the histogram.

> **N**ote: In both implementation, there is no protection against illegal arguments. Illegal arguments can occur when computing the distribution value when the histogram underflow and overflow counts are non zero. Below the minimum and above the maximum, no information can be obtained for the distribution. Within the histogram limits, there is no need for protection. Therefore the implementation assumes that the histogram was collected using automatic adjustment of the limits (*c.f.* section 9.3).

Experimental distribution — Smalltalk implementation

Listing **??** shows the implementation of an experimental distribution in Smalltalk.

The class PMHistogrammedDistribution is a subclass of the class PMProbabilityDensity. The class creation method histogram: takes as argument the histogram over which the instance is defined. To prevent creating a instance with undefined instance variable, the default class creation method new returns an error.

Listing 9-20 m

```
alltalk implementation of an experimental distribution
    \label{ls:histprob}
\input{Smalltalk/Statistics/DhbHistogrammedDistribution}
```

CHAPTER 10

Statistical analysis

L'expérience instruit plus sûrement que le conseil.[1]
André Gide

This chapter is dedicated on how to extract information from large amount of data using statistical analysis. One of the best book I have read on this subject is titled *H*ow to lies with statistics[2]. This admittedly slanted title seems a little pessimistic. The truth, however, is that most people in their daily job ignore the little statistics they have learned in high school. As a result, statistical argumentation is often used wrongly to produce the wrong conclusions.

The problems addressed in this section pertain to the interpretation of experimental measurements. For a long time such a discipline was reserved to physicists only. Recently natural science disciplines discovered that statistics could be used effectively to verify hypotheses or to determine parameters based on experimental observations. Today, the best papers on statistics and estimations are found primarily in natural science publications (Biometrika *e.g.*). Recently, the use of statistics has been extended to financial analysis.

Statistics can be applied to experimental data in two ways. Firstly, one can test the consistency and/or accuracy of the data. These tests are the subject of the first 3 sections. Secondly, the values of unknown parameters can be derived from experimental data. This very important aspect of statistical analysis is treated in the remaining of the chapter.

Figure 10-1 shows the classes described in this chapter. The reader should be aware that the techniques used for non-linear least square fits (section 10.9) can also be also applied to solve systems of non-linear equations.

[1] Experience teaches more surely than counseling.
[2] D. Huff, *H*ow to lies with statistics, Norton and Co., New York 1954.

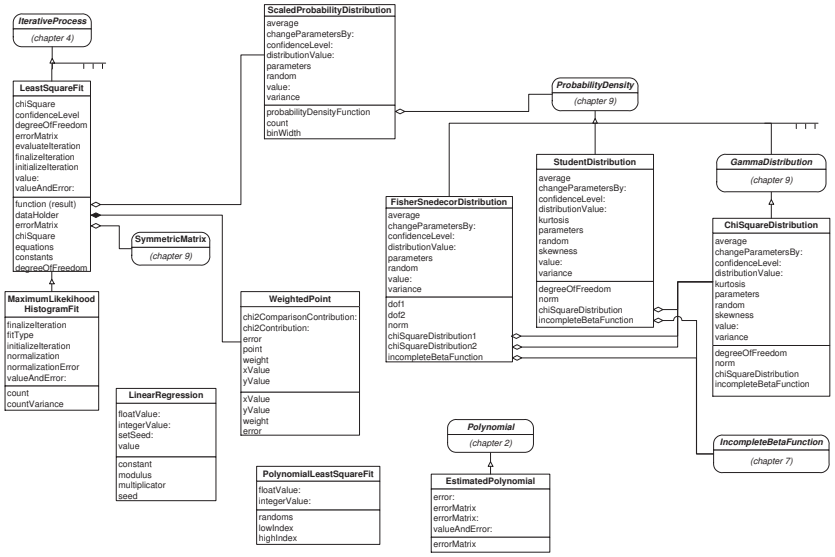


Figure 10-1 Classes related to estimation

10.1 F-test and the Fisher-Snedecor distribution

The F-test tries to answer the following question: given two series of measurements, $x_1, \ldots, x_n$ and $y_1, \ldots, y_m$, what is the probability that the two measurements have the same standard deviation? The F-test is used when there are not enough statistics to perform a detailed analysis of the data.

Let us assume that the distribution of the two random variables, x and y, are normal distributions with respective averages μ_x and μ_y, and respective standard deviations σ_x and σ_y. Then, $\bar{s}_x$, the standard deviation of $x_1, \ldots, x_n$ is an estimator of σ_x; $\bar{s}_y$, the standard deviation of $y_1, \ldots, y_m$ is an estimator of σ_y. The following statistics

$$F = \frac{\bar{s}_x^2}{\sigma_x^2} \cdot \frac{\sigma_y^2}{\bar{s}_y^2} \tag{10.1}$$

can be shown to be distributed according to a Fisher-Snedecor distribution with degrees of freedom n and m. In particular, if one wants to test for the equality of the two standard deviations, one construct the following statistics:

$$F = \frac{\bar{s}_x^2}{\bar{s}_y^2}. \tag{10.2}$$

Traditionally one chooses $\bar{s}_x > \bar{s}_y$ so that the variable F is always greater than one.

It is important to recall that the expression above is distributed according to a Fisher-Snedecor distribution if and only if the two sets of data are dis-

tributed according to a normal distribution. For experimental measurements this is often the case unless systematic errors are present. Nevertheless this assumption must be verified before making an F-test.

Table 10-2 shows the properties of the Fisher-Snedecor distribution. The Fisher-Snedecor distribution is itself rarely used as a probability density function, however.

Table 10-2 Properties of the Fisher-Snedecor distribution

Range of random variable	$[0, +\infty[$
Probability density function	$P(x) = \dfrac{n_1^{\frac{n_1}{2}} n_2^{\frac{n_2}{2}} x^{\frac{n_1-1}{2}}}{B\left(\frac{n_1}{2}, \frac{n_2}{2}\right)(n_1 + n_2 x)^{\frac{n_1+n_2}{2}}}$
Parameters	n_1, n_2 two positive integers
Distribution function	$F(x) = B\left(\dfrac{n_1}{n_1 + n_2 x}; \dfrac{n_1}{2}, \dfrac{n_2}{2}\right)$
Average	$\dfrac{n_2}{n_2 - 2}$ for $n > 2$ undefined otherwise
Variance	$\dfrac{2n_2^2(n_1 + n_2 - 2)}{n_1(n_2 - 2)^2(n_2 - 4)}$ for $n > 4$ undefined otherwise
Skewness	
Kurtosis	

The main part of figure 10-3 shows the shape of the Fisher-Snedecor distribution for some values of the degrees of freedom. For large n and m, the Fisher-Snedecor distribution tends toward a normal distribution.

The confidence level of a Fisher-Snedecor distribution is defined as the probability expressed in percent of finding a value larger than F defined in equation 10.2 or lower than $1/F$. The confidence level is thus related to the distribution function. We have:

$$\mathrm{CL}_F(x, n) = 100\left\{1 - \left[F(x) - F\left(\frac{1}{x}\right)\right]\right\}, \tag{10.3}$$

where x is the value F defined in equation 10.2. The change of notation was made to avoid confusion between the acceptance function and the random variable. The confidence level corresponds to the surface of the shaded areas in the insert in the upper right corner of figure 10-3.

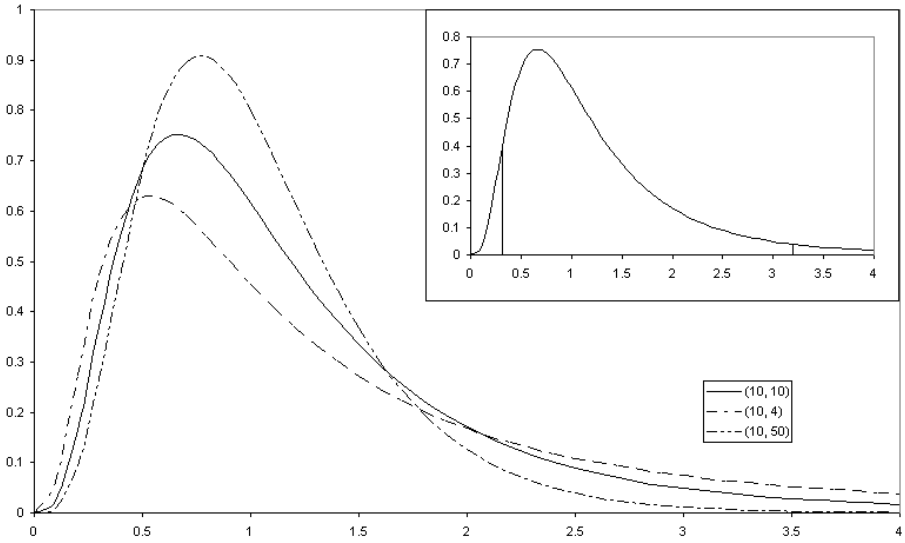


Figure 10-3 Fisher-Snedecor distribution for a few parameters

Example.

Here is an example used to investigate the goodness of the two random number generators discussed in section 9.4. The collected data are the error of the covariance test described in section 9.4. The dimension of the covariance matrix is 7 and the number of trials on each measurement was 1000. The table 10-4 presents the obtained results, expressed in $\frac{1}{1000}$. the ratio of the two

Table 10-4 Covariance test of random number generator

	Congruential	Mitchell-Moore
1	5.56	7.48
2	5.89	6.75
3	4.66	3.77
4	5.69	5.71
5	5.34	7.25
6	4.79	4.73
7	4.80	6.23
8	7.86	5.60
9	3.64	5.94
10	5.70	4.58
Average	5.40	5.80
Std. dev.	1.10	1.19

variances is 1.18 and the degrees of freedom are both 10. The F-test applied to the two variances gives a confidence level of 20%. This means that there is

only a 20% probability that the variances of the two series of measurements are different.

Fisher-Snedecor distribution — Smalltalk implementation

Figure 10-1 with the box FisherSnede grayed.

Listing 10-5 shows the implementation of the Fisher-Snedecor distribution in Smalltalk. Listing 10-6 shows the implementation of the F-test in Smalltalk. The following code example shows how to perform a F-test between two sets of experimental measurements.

```
| mom1 mom2 confidenceLevel |
mom1 := PMFixedStatisticalMoments new.
```

<Collecting measurements of set 1 into mom1>

```
mom2 := PMFixedStatisticalMoments new.
```

<Collecting measurements of set 2 into mom1>

```
confidenceLevel := mom1 fConfidenceLevel: mom2.
```

Two instances of statistical moments (*c.f.* section 9.2) are created. Experimental data are accumulated into each set separately (*c.f.* code example **??**). The last line returns the probability in percent that the two sets of data have the same standard deviation.

The class PMFisherSnedecorDistribution is implemented as a subclass of PMProbabilityDensity because its distribution function can be computed numerically using the incomplete beta function (*c.f.* section 7.5).

Listing 10-5 Smalltalk implementation of the Fisher-Snedecor distribution

```
PMProbabilityDensity subclass: #PMFisherSnedecorDistribution
  instanceVariableNames: 'dof1 dof2 norm chiSquareDistribution1
  chiSquareDistribution2 incompleteBetaFunction'
  classVariableNames: ''
  package: 'Math-DistributionForHistogram'
```

```
PMFisherSnedecorDistribution class >> degreeOfFreedom: anInteger1
degreeOfFreedom: anInteger2
    ^ super new initialize: anInteger1 and: anInteger2
```

```
PMFisherSnedecorDistribution class >> distributionName
    ^ 'Fisher-Snedecor distribution'
```

```
PMFisherSnedecorDistribution class >> fromHistogram: aHistogram
    | n1 n2 a |
    aHistogram minimum < 0 ifTrue: [^nil].
    n2 := (2 / (1 - (1 / aHistogram average))) rounded.
    n2 > 0 ifFalse: [^nil].
    a := (n2 - 2) * (n2 - 4) * aHistogram variance / (n2 squared *

    2).
```

```
    n1 := (0.7 * (n2 - 2) / (1 - a)) rounded.
    ^n1 > 0
        ifTrue: [self degreeOfFreedom: n1 degreeOfFreedom: n2]
        ifFalse: [nil]
```

```
PMFisherSnedecorDistribution class >> new
 ^ self error: 'Illegal creation message for this class'
```

```
PMFisherSnedecorDistribution class >> test: aStatisticalMoment1
    with: aStatisticalMoment2
    ^ (self class degreeOfFreedom: aStatisticalMoment1 count
        degreeOfFreedom: aStatisticalMoment2 count)
            distributionValue: aStatisticalMoment1 variance
                    / aStatisticalMoment2 variance
```

```
PMFisherSnedecorDistribution >> average
    ^dof2 > 2
        ifTrue: [ dof2 / ( dof2 - 2) ]
        ifFalse:[ nil ]
```

```
PMFisherSnedecorDistribution >> changeParametersBy: aVector
    dof1 := (dof1 + (aVector at: 1)) max: 1.
    dof2 := (dof2 + (aVector at: 2)) max: 1.
    self computeNorm.
    chiSquareDistribution1 := nil.
    chiSquareDistribution2 := nil.
    incompleteBetaFunction := nil
```

```
PMFisherSnedecorDistribution >> computeNorm
    norm := (dof1 ln * (dof1 / 2)) + (dof2 ln * (dof2 / 2))
                        - ((dof1 / 2) logBeta: (dof2 / 2))
```

```
PMFisherSnedecorDistribution >> confidenceLevel: aNumber
    aNumber < 0
        ifTrue: [ self error: 'Confidence level argument must be

    positive'].
    ^((self distributionValue: aNumber) - ( self distributionValue:
                                          aNumber reciprocal) ) *
    100
```

```
PMFisherSnedecorDistribution >> distributionValue: aNumber
    ^ 1 - ( self incompleteBetaFunction value: ( dof2 / ( aNumber *
                                                          dof1 +
    dof2)))
```

```
PMFisherSnedecorDistribution >> incompleteBetaFunction
    incompleteBetaFunction isNil
        ifTrue:
            [incompleteBetaFunction := PMIncompleteBetaFunction
                                                        shape: dof2 /
    2
                        shape: dof1 / 2].
```

```
    ^incompleteBetaFunction

PMFisherSnedecorDistribution >> initialize: anInteger1 and:
    anInteger2
    dof1 := anInteger1.
    dof2 := anInteger2.
    self computeNorm.
    ^ self

PMFisherSnedecorDistribution >> parameters
    ^ Array with: dof1 with: dof2

PMFisherSnedecorDistribution >> random
    chiSquareDistribution1 isNil
        ifTrue: [ chiSquareDistribution1 := PMChiSquareDistribution
                                                degreeOfFreedom:
    dof1.
                  chiSquareDistribution2 := PMChiSquareDistribution
                                                degreeOfFreedom:
    dof2.
                ].
    ^chiSquareDistribution1 random * dof2 / ( chiSquareDistribution2
                                                        random *
    dof1)

PMFisherSnedecorDistribution >> value: aNumber
    ^aNumber > 0
        ifTrue: [ (norm + ( aNumber ln * (dof1 / 2 - 1)) - (
             (aNumber * dof1 + dof2) ln * (( dof1 + dof2) / 2))) exp]
        ifFalse:[ 0 ]

PMFisherSnedecorDistribution >> variance
    ^ dof2 > 4 ifTrue: [ dof2 squared * 2 * ( dof1 + dof2 - 2) / ( (
                              dof2 - 2) squared * dof1 * ( dof2 -
    4))]
                   ifFalse:[ nil ]
```

The computation of the confidence level for the F-test is implemented in the method `fConfidenceLevel:` of the class `PMStatisticalMoments`. It calculates the statistics F according to equation 10.2, creates an instance of a Fisher-Snedecor distribution and passes the value of F to the method `confidenceLevel:` of the distribution. The method `fConfidenceLevel:` is also implemented by the class `Histogram` where it is simply delegated to the statistical moments accumulated by the histogram. The argument of the method can be a statistical moment or a histogram since the messages sent by the method are polymorphic to both classes.

Listing 10-6 Smalltalk implementation of the F-test

```
Object subclass: #PMStatisticalMoments
  instanceVariableNames: 'moments'
  classVariableNames: ''
  package: 'Math-StatisticalMoments'
```

```
PMStatisticalMoments >> fConfidenceLevel:
    aStatisticalMomentsOrHistogram
    | fValue |
    fValue := self variance/ aStatisticalMomentsOrHistogram variance.
    ^ fValue < 1
        ifTrue: [ (PMFisherSnedecorDistribution degreeOfFreedom:
                                    aStatisticalMomentsOrHistogram
    count
                        degreeOfFreedom: self count)
                                    confidenceLevel: fValue

    reciprocal ]
        ifFalse: [ (PMFisherSnedecorDistribution degreeOfFreedom:
                                                            self
    count
                        degreeOfFreedom:
                                aStatisticalMomentsOrHistogram count)
                                    confidenceLevel: fValue ]
```

```
Object subclass: #PMHistogram
instanceVariableNames: 'minimum binWidth overflow underflow moments
    contents freeExtent cacheSize desiredNumberOfBins'
classVariableNames: ''
package: 'Math-StatisticalMoments'
```

```
PMHistogram >> fConfidenceLevel: aStatisticalMomentsOrHistogram
    ^ moments fConfidenceLevel: aStatisticalMomentsOrHistogram
```

10.2 t-test and the Student distribution

The t-test tries to answer the following question: given two series of measurements, $x_1, \ldots, x_n$ and $y_1, \ldots, y_m$, what is the probability that the two measurements have the same average? The t-test is used when there are not enough statistics to perform a detailed analysis of the data.

Let us assume that the distribution of the two random variables, x and y, are normal distributions with respective averages μ_x and μ_y, and the same standard deviation σ. Then $\bar{x}$, the average of $x_1, \ldots, x_n$ is an estimator of μ_x; $\bar{y}$, the average of $y_1, \ldots, y_m$ is an estimator of μ_y. An estimation $\bar{s}$ of the standard deviation σ can be made using both measurement samples. We have:

$$\bar{s}^2 = \frac{\sum_{i=1}^{n} (x_i - \bar{x})^2 + \sum_{i=1}^{m} (y_i - \bar{y})^2}{n + m - 2}. \tag{10.4}$$

One can prove that the following statistics:

$$t = \frac{(\bar{x} - \bar{y}) - (\mu_x - \mu_y)}{\bar{s}\sqrt{\frac{1}{n} + \frac{1}{m}}} \tag{10.5}$$

is distributed according to a Student distribution with $n + m - 2$ degrees of freedom. In particular, to test for the probability that the two series of measurements have the same average, one uses the following statistics:

$$t = \frac{\bar{x} - \bar{y}}{\bar{s}\sqrt{\frac{1}{n} + \frac{1}{m}}}. \tag{10.6}$$

It is important to recall the two fundamental hypotheses that have been made so far.

1. The two sets of data must be distributed according to a normal distribution.
2. The two sets of data must have the same standard deviation.

Too many people use the t-test without first checking the assumptions. Assumption 1 is usually fulfilled with experimental measurements in the absence of systematic errors. Assumption 2. however, must be checked, for example using the F-test discussed in section 10.1.

Because the random variable of the distribution is traditionally labeled t, this distribution is often called the t-distribution. Table 10-7 shows the properties of the Student distribution. The Student distribution is itself rarely used as a probability density function, however.

For $n = 1$, the Student distribution is identical to a Cauchy distribution with $\mu = 0$ and $\beta = 1$. For large n, the Student distribution tends toward a normal distribution with average 0 and variance 1. The main part of figure 10-8 shows the shapes of the Student distribution for a few values of the degrees of freedom. The normal distribution is also given for comparison.

The confidence level of a Student distribution is defined as the probability to find a value whose absolute value is larger than a given value. Thus, it estimates the level of confidence that the hypothesis — namely, that the two sets of measurements have the same average — cannot be accepted. Traditionally the confidence level is given in percent. The confidence level corresponds to the surface of shaded area in the insert in the upper left corner of figure 10-8. By definition, the confidence level is related to the interval acceptance function:

$$\mathrm{CL}_t\left(t, n\right) = 100\left[1 - F\left(-\left|t\right|, \left|t\right|\right)\right], \tag{10.7}$$

using the definition of the interval acceptance function (equation 9.31). The value of t in equation 10.7 is obtained from equation 10.6.

The distribution function of the Student distribution is calculated with the incomplete beta function (*c.f.* section 7.5). Using the fact that the distribution is symmetric, one can derive the following expression

$$F\left(-\left|t\right|, \left|t\right|\right) = B\left(\frac{n}{n + t^2}; \frac{n}{2}, \frac{1}{2}\right), \tag{10.8}$$

Table 10-7 Properties of the Student distribution

Range of random variable	$]-\infty, +\infty[$
Probability density function	$P(x) = \frac{1}{\sqrt{n}B\left(\frac{n}{2}, \frac{1}{2}\right)} \left(1 + \frac{t^2}{n}\right)^{-\frac{n+1}{2}}$
Parameters	n a positive integer
Distribution function	$F(x) = \begin{cases} \frac{1+B\left(\frac{n}{n+x^2}; \frac{n}{2}, \frac{1}{2}\right)}{2} & \text{for } x \geq 0 \\ \frac{1-B\left(\frac{n}{n+x^2}; \frac{n}{2}, \frac{1}{2}\right)}{2} & \text{for } x < 0 \end{cases}$
Average	0
Variance	$\frac{n}{n-2}$ for $n > 2$ undefined otherwise
Skewness	0
Kurtosis	$\frac{6}{n-4}$ for $n > 4$ undefined otherwise

from the properties of the distribution (*c.f.* table 10-7) and using equations 10.7 and 7.12.

Example

Now, we shall continue the analysis of the results of table 10-4. The t value computed from the two sets of measurements is 0.112 for a degree of freedom of 18. the corresponding confidence level is 8.76%. That is, there is only a 8.76% probability that the two generators have a different behavior. Thus, we can conclude that the Mitchell-Moore random generator is as good as the congruential random generator.

Student distribution — Smalltalk implementation

Figure 10-1 with the box StudentDistribution grayed.

Listing **??** shows the implementation of the Student distribution in Smalltalk. Listing **??** shows the implementation of the t-test in Smalltalk. Performing a t-test between two sets of experimental measurements is very similar to performing a F-test. In code example 10.1 it suffices to replace the last line with the following:

```
onfidenceLevel := mom1 fConfidenceLevel: mom2.
```

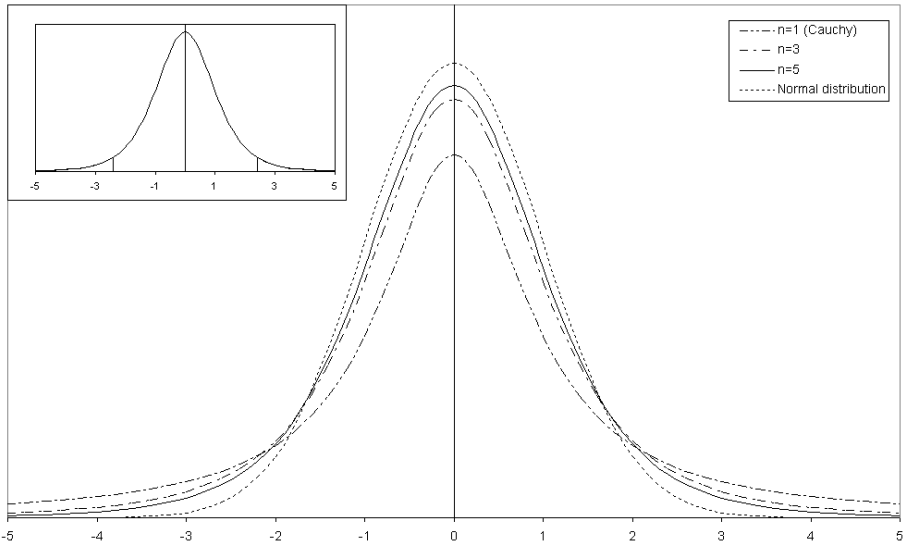


Figure 10-8 Student distribution for a few degrees of freedom

This last line returns the probability in percent that the two sets of data have the same average provided that the two sets have the same standard deviation.

The class PMStudentDistribution is implemented as a subclass of PMProbabilityDensity because its distribution function can be computed numerically using the incomplete beta function (*c.f.* section 7.5).

The method symmetricAcceptance: computes the symmetric acceptance function defined by equation 10.8. This method is used to compute the disribution function and the confidence level. The method confidenceLevel: gives the confidence level in percent.

Listing 10-9 m

```
alltalk implementation of the Student distribution \label{ls:tdist}
\input{Smalltalk/Statistics/DhbStudentDistribution}
```

The computation of the confidence level for the t-test is implemented in the method tConfidenceLevel: of the class PMStatisticalMoments. It calculates the statistics t according to equation 10.6, creates an instance of a Student distribution and passes the value of t to the method confidenceLevel: of the distribution. The method tConfidenceLevel: is also implemented by the class Histogram where it is simply delegated to the statistical moments accumulated by the histogram. The argument of the method can be a statistical moment or a histogram since the messages sent by the method are polymorphic to both classes.

The method unnormalizedVariance of class PMStatisticalMoments corresponds to each sums in the numerator of equation 10.4. To allow performing a t-test also with instances of class PMFastStatisticalMoments, it was necessary to define this for that class.

Listing 10-10 m

```
alltalk implementation of the $t$-test \label{ls:tTest}
\input{Smalltalk/Estimation/DhbStatisticalMoments(DhbEstimation)}
\input{Smalltalk/Estimation/DhbFastStatisticalMoments(DhbEstimation)}
\input{Smalltalk/Estimation/DhbHistogram(DhbEstimation)}
```

10.3 χ^2-test and χ^2 distribution

The χ^2-test tries to answer the following question: how well a theory is able to predict observed results? Alternatively a χ^2-test can also tell whether two independent sets of observed results are compatible. This latter formulation is less frequently used than the former. Admittedly these two questions are somewhat vague. We shall now put them in mathematical terms for a more precise definition.

Let us assume that the measurement of an observable quantity depends on some parameters. These parameters cannot be adjusted by the experimenter but can be measured exactly[3]. Let x_p be the measured values of the observed quantity where p is a label for the parameters; let σ_p be the standard deviation of x_p.

The first question assumes that one can predict the values of the observed quantity: let μ_p be the predicted value of x_p. Then the quantity:

$$y_p = \frac{x_p - \mu_p}{\sigma_p} \tag{10.9}$$

is distributed according to a normal distribution with average 0 and standard deviation 1 if and only if the quantities x_p are distributed according to a normal distribution with average μ_p and standard deviation σ_p.

A χ^2 distribution with n degrees of freedom describes the distribution of the sum of the squares of n random variables distributed according to a normal distribution with mean 0 and standard deviation 1. Thus, the following quantity

$$S = \sum_p \frac{(x_p - \mu_p)^2}{\sigma_p^2} \tag{10.10}$$

is distributed according to a χ^2 distribution with n degrees of freedom where n is the number of available measurements (that is the number of terms in the sum of equation 10.10).

[3] Of course, there is no such thing as an exact measurement. The measurement of the parameters must be far more precise than that of the observed quantity.

To formulate the second question one must introduce a second set of measurement of the same quantity and at the same values of the parameters. Let x'_p be the second set of measured values and σ'_p the corresponding standard deviations. The estimated standard deviation for the difference $x_p - x'_p$ is $\sqrt{\sigma_p^2 + \sigma_p'^2}$. If the two sets of measurements are compatible, the quantity

$$y'_p = \frac{x_p - x'_p}{\sqrt{\sigma_p^2 + \sigma_p'^2}} \tag{10.11}$$

is distributed according to a normal distribution with average 0 and standard deviation 1. Then the following quantity

$$S = \sum_p \frac{\left(x_p - x'_p\right)^2}{\sigma_p^2 + \sigma_p'^2 2} \tag{10.12}$$

is distributed according to a χ^2 distribution with n degrees of freedom.

Table 10-11 shows the properties of the χ^2 distribution. It is a special case of

Table 10-11 Properties of the χ^2 distribution

Range of random variable	$[0, +\infty[$
Probability density function	$P(x) = \frac{x^{\frac{n}{2}-1} e^{-\frac{x}{2}}}{2^{\frac{n}{2}} \Gamma\left(\frac{n}{2}\right)}$
Parameters	n a positive integer
Distribution function	$F(x) = \Gamma\left(\frac{x}{2}; \frac{n}{2}\right)$
Average	n
Variance	$2n$
Skewness	$2\sqrt{\frac{2}{n}}$
Kurtosis	$\frac{12}{n}$

the gamma distribution with $\alpha = \frac{n}{2}$ and $\beta = 2$ (*c.f.* section 9.7). For $n > 30$ one can prove that the variable $y = \sqrt{2x} - \sqrt{2n-1}$ is approximately distributed according to a normal distribution with average 0 and standard deviation 1. If n is very large the χ^2 distribution tends toward a normal distribution with average n and standard deviation $\sqrt{2n}$.

Figure 10-12 shows the shape of the χ^2 distribution for a few values of the degree of freedom.

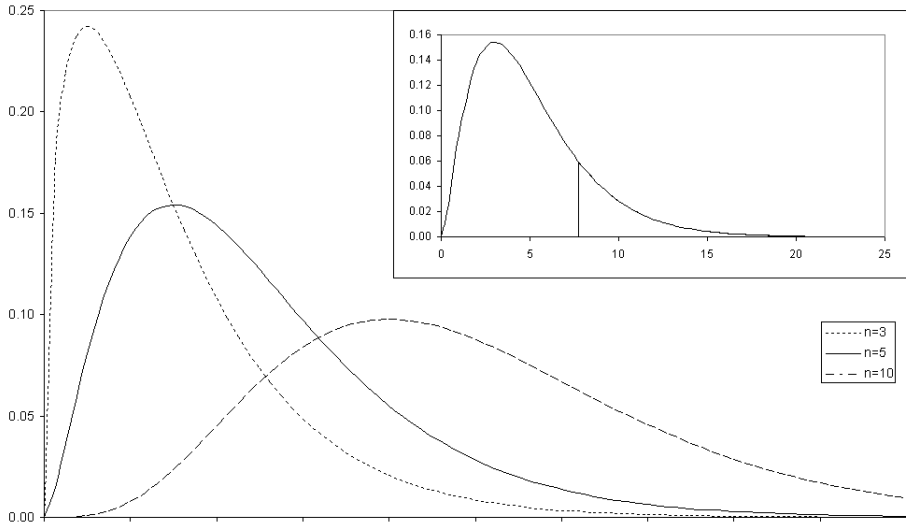


Figure 10-12 χ^2 distribution for a few degrees of freedom

To perform a χ^2-test, it is customary to evaluate the probability of finding a value larger than the value obtained in equations 10.10 or 10.12. In this form, the result of a χ^2-test gives the probability that the set of measurements is *n*ot compatible with the prediction or with another set of measurements. The confidence level of a χ^2 value is defined as the probability of finding a value larger than χ^2 expressed in percent. It is thus related to the distribution function as follows:

$$\mathrm{CL}_S = 100\left[1 - F\left(S\right)\right], \tag{10.13}$$

where S is the quantity defined in equation 10.10 or 10.12. The confidence level corresponds to the surface of the shaded area of the insert in the upper right corner of figure 10-12.

Since the χ^2 distribution is a special case of the gamma distribution the confidence level can be expressed with the incomplete gamma function (*c.f.* section 7.4):

$$\mathrm{CL}_S = 100\left[1 - \Gamma\left(\frac{S}{2}, \frac{n}{2}\right)\right]. \tag{10.14}$$

For large n the χ^2 confidence level can be computed from the error function (*c.f.* section 2.4):

$$\mathrm{CL}_S = 100\left[1 - \mathrm{erf}\left(\sqrt{2S} - \sqrt{2n-1}\right)\right]. \tag{10.15}$$

Figure 10-1 with the box ChiSquaredDistribution grayed.

χ^2 distribution — Smalltalk implementation

Listing **??** shows the implementation of the χ^2 distribution in Smalltalk. The asymptotic limit is implemented directly in the class creation method.

Listing 10-13 m

```
alltalk implementation of the $\chi^2$ distribution
    \label{ls:chidist}
\input{Smalltalk/Statistics/DhbChiSquareDistribution}
```

Weighted point implementation

Figure 10-1 with the box WeightedPo grayed.

As we shall see in the rest of this chapter, the evaluation of equation 10.10 is performed at many places. Thus, it is convenient to create a new class handling this type of calculation. The new class is called PMWeightedPoint in Smalltalk.

A weighted point has the following instance variables:

xValue the x value of the data point, that is x_i,

yValue the y value of the data point, that is y_i,

weight the weight of the point, that is $1/\sigma_i^2$ and

error the error of the y value, that is σ_i.

Accessor methods for each of these instance variables are provided. The accessor method for the error is using lazy initialization to compute the error from the weight in case the error has not yet been defined.

The method chi2Contribution — with an added semicolon at the end of the name for Smalltalk — implements the computation of one term of the sum in equation 10.10. The argument of the method is any object implementing the behavior of a one-variable function defined in section 2.1.

Creating instances of the classes can be done in many ways. The fundamental method takes as arguments x_i, y_i and the weight $1/\sigma_i^2$. However convenience methods are provided for frequent cases:

1. x_i, y_i and the error on y_i, σ_i;
2. x_i and the content of a histogram bin; the weight is derived from the bin contents as explained in section 10.4;
3. x_i, y_i without known error; the weight of the point is set to 1; points without error should not be used together with points with errors;
4. x_i and a statistical moment; in this case, the value y_i is an average over a set of measurements; the weight is determined from the error on the average (*c.f.* section 9.1);

Examples of use of weighted points appear in many sections of this chapter (10.4, 10.8, 10.9).

In the Smalltalk class PMWeightedPoint the values x_i and y_i are always supplied as an instance of the class Point. The class PMWeightedPoint has the following class creation methods:

point:weight: fundamental method;

point:error: convenience method 1;

point:count: convenience method 2;

point: convenience method 3

The convenience method 4 is implemented by the method asWeightedPoint of the class PMStatisticalMoments. This kind of technique is quite common in Smalltalk instead of making a class creation method with an explicit name (fromMoment: *e.g.*).

Listing 10-14 m

```
alltalk implementation of the weighted point class
\label{ls:weightedPoint}
\input{Smalltalk/Estimation/DhbWeightedPoint}
\input{Smalltalk/Estimation/DhbStatisticalMoments(DhbEstimationChi2)}
```

10.4 χ^2-test on histograms

As we have seen in section 9.3 histograms are often used to collect experimental data. Performing a χ^2-test of data accumulated into a histogram against a function is a frequent task of data analysis.

The χ^2 statistics defined by equation 10.10 requires an estimate of the standard deviation of the content of each bin. One can show that the contents of a histogram bin is distributed according to a Poisson distribution. The Poisson distribution is a discrete distribution[4] whose average is equal to the variance. The probability of observing the integer k is defined by:

$$P_\mu(k) = \frac{\mu^k}{k!} e^\mu, \tag{10.16}$$

where μ is the average of the distribution. In the case of a histogram, the estimated variance of the bin content is then the bin content itself. Therefore equation 10.10 becomes:

$$S = \sum_{i=1}^{n} \frac{\left\{n_i - \mu\left[x_{\min} + \left(i + \frac{1}{2}\right) w\right]\right\}^2}{n_i}, \tag{10.17}$$

where n is the number of bins of the histogram, $x_{\min}$ its minimum and w its bin width. The estimation of the bin content against which the χ^2 statistics is

[4] A discrete distribution is a probability distribution whose random variable is an integer.

computed, μ, is now a function evaluated at the middle of each bin to average out variations of the function over the bin interval.

In fact, the function μ is often related to a probability density function since histograms are measuring probability distributions. In this case the evaluation is somewhat different. Let $P(x)$ be the probability density function against which the χ^2 statistics is computed. Then, the predicted bin content for bin i is given by:

$$\mu_i = wNP\left[x_{\text{min}} + \left(i + \frac{1}{2}\right)w\right], \tag{10.18}$$

where N is the total number of values accumulated in the histogram. This is a symmetric version of the definition of a probability density function: w plays the role of dx in equation 9.23. Plugging equation 10.18 into equation 10.17 yields the expression of the χ^2 statistics for a histogram computed against a probability density function $P(x)$:

$$S = \sum_{i=1}^{n} \frac{\left\{n_i - wNP\left[x_{\text{min}} + \left(i + \frac{1}{2}\right)w\right]\right\}^2}{n_i}, \tag{10.19}$$

This equation cannot be applied for empty bins. If the bin is empty one can set the weight to 1. This corresponds to a 63% probability of observing no counts if the expected number of measurement is larger than 0.

In both implementations a single class is in charge of evaluating the predicted bin contents. This class is called a scaled probability density function. It is defined by a probability distribution and a histogram.

χ^2-test on histograms — Smalltalk implementation

Figure 10-1 with the box ScaledProba grayed.

Listing **??** shows the implementation of a scaled probability density function in Smalltalk. Listing **??** shows the additional methods for the class PMHistogram needed to perform a χ^2-test. Examples of use are given in sections 10.9 and 10.10. Here is a simple example showing how to compute a χ^2-confidence level to estimate the goodness of a random number generator.

```
\label{exs:chitest}
 | trials probDistr histogram |
 trials := 5000.
 probDistr := PMNormalDistribution new.
 histogram := PMHistogram new.
 histogram freeExtent: true; setDesiredNumberOfBins: 100.
 trials timesRepeat: [ histogram accumulate: probDistr random ].
 histogram chi2ConfidenceLevelAgainst:
        ( PMScaledProbabilityDensityFunction histogram: histogram
                                             against: probDistr)
```

The first line after the declaration defines the number of data to be generated to 5000. After, an new instance of a probability distribution — in this case a normal distribution with average 0 and variance 1 — is created. Then, a new instance of a histogram is created and the next line defines it with a rough number of bins of 100 and the ability to automatically adjust its limits. After all instances have been created, random data generated by the probability distribution are generated. The last statement — extending itself over the last three lines — calculates the confidence level. The argument of the method chi2ConfidenceLevelAgainst: is a scaled probability distribution constructed over the histogram and the probability distribution used to generate the accumulated data.

The class PMScaledProbabilityDensityFunction has two class creation methods. The class method histogram:against: takes two arguments, a histogram and a probability distribution. This method is used to perform a χ^2-test of the specified histogram against the given probability distribution. The class method histogram:distributionClass: first create a probability distribution of the given class using parameters estimated from the histogram. This method is used to create a scaled probability density function whose parameters will be determined with least square or maximum likelihood fits.

Listing 10-15 m

```
alltalk implementation of a scaled
probability density function \label{ls:scaleddist}
\input{Smalltalk/Statistics/DhbScaledProbabilityDensityFunction}
```

The evaluation of equation 10.19 is performed by the method chi2Against: of the class PMHistogram. This method uses the iterator method pointsAndErrorsDo:. This method iterates on all bins and performs on each of them a block using as argument a weighted point as described in section 10.3. This iterator method is also used for least square and maximum likelihood fits (*c.f.* sections 10.9 and 10.10).

Listing 10-16 m

```
alltalk implementation of $\chi^2$-test on histograms
    \label{ls:chitesthist}
\input{Smalltalk/Estimation/DhbHistogram(DhbEstimationChi2)}
```

10.5 Definition of estimation

Let us assume that an observable quantity y is following a probability distribution described by a set of observable quantities $x_1, x_2 \ldots$ - called the experimental conditions - and a set of parameters $p_1, p_2 \ldots$. In other words, the robability density function of the random variable[5] corresponding to the

[5] For simplicity we shall use the same notation for the random variable and the observable quantity.

observable quantity y can be written as

$$P(y) = P(y; \mathbf{x}, \mathbf{p}), \tag{10.20}$$

where $\mathbf{x}$ is the vector $(x_1, x_2 \ldots)$ and $\mathbf{p}$ the vector $(p_1, p_2 \ldots)$.

The estimation of the values of the parameters $p_1, p_2 \ldots$ is the determination of the parameters $p_1, p_2 \ldots$ by performing several measurements of the observable y for different experimental conditions $x_1, x_2 \ldots$.

Let N be the number of measurements; let y_i be the i^{th} measured value of the observable y under the experimental conditions $\mathbf{x}_i$.

Maximum likelihood estimation

The maximum likelihood estimation of the parameters $\mathbf{p}$ is the set of values $\bar{\mathbf{p}}$ maximizing the following function:

$$L(\mathbf{p}) = \prod_{i=1}^{N} P(y_i; \mathbf{x}_i, \mathbf{p}) \tag{10.21}$$

By definition the likelihood function $L(\mathbf{p})$ is the probability of making the N measurements. The maximum likelihood estimation determines the estimation $\bar{\mathbf{p}}$ of the parameters $\mathbf{p}$ such that the series of measurements performed is the most probable, hence the name maximum likelihood.

One can show that the maximum likelihood estimation is robust and unbiased. The robustness and the bias of an estimation are defined mathematically. For short, *r*obust means that the estimation converges toward the true value of the parameters for an infinite number of measurements; *u*nbiased means that the deviations between the estimated parameters and their true value are symmetrically distributed around 0 for any finite number of measurements.

Equation 10.21 is often rewritten in logarithmic form to ease the computation of the likelihood function.

$$I(\mathbf{p}) = \ln L(\mathbf{p}) = \sum_{i=1}^{N} \ln P(y_i; \mathbf{x}_i, \mathbf{p}) \tag{10.22}$$

The function $I(\mathbf{p})$ is related to information and is used in information theory.

Least square estimation

Let us assume that the random variable y is distributed according to a normal distribution of given standard deviation σ and that the average of the normal

distribution is given by a function $F(\mathbf{x}, \mathbf{p})$ of the experimental conditions and the parameters. In this case equation 10.20 becomes:

$$P(y) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{\left[y - F\left(\mathbf{x}, \mathbf{p}\right)\right]^2}{2\sigma^2}} \tag{10.23}$$

Plugging equation 10.23 into equation 10.22 yields:

$$I(\mathbf{p}) = -N\sqrt{2\pi\sigma^2} - \sum_{i=1}^{N} \frac{\left[y - F(\mathbf{x}, \mathbf{p})\right]^2}{2\sigma^2} \tag{10.24}$$

The problem of finding the maximum of $I(\mathbf{p})$ is now equivalent to the problem of finding the minimum of the function:

$$S_{ML}(\mathbf{p}) = \sum_{i=1}^{N} \frac{\left[y - F(\mathbf{x}, \mathbf{p})\right]^2}{\sigma^2}, \tag{10.25}$$

where a redundant factor 2 has been removed. This kind of estimation is called least square estimation. Written as in equation 10.25 least square estimation is fully equivalent to maximum likelihood estimation. By definition, the quantity $S_{ML}(\mathbf{p})$ is distributed as a χ^2 random variable with $N - m$ degrees of freedom where m is the number of parameters, that is the dimension of the vector $\mathbf{p}$.

In practice, however, the standard deviation σ is not known and frequently depends on the parameters $\mathbf{p}$. In that case, one uses instead an estimation for the standard deviation. Either the standard deviation of each measurement is determined experimentally by making several measurements under the same experimental conditions or it is estimated from the measurement error. Then, equation 10.25 can be rewritten as:

$$S(\mathbf{p}) = \sum_{i=1}^{N} \frac{\left[y - F(\mathbf{x}, \mathbf{p})\right]^2}{\sigma_i^2}. \tag{10.26}$$

The least square estimation is obtained by minimizing the quantity $S(\mathbf{p})$ with respect to the parameters $\mathbf{p}$. This kind of estimation can be used to determine the parameters of a functional dependence of the variable y from the observable quantities $\mathbf{x}$. For this reason it is also called a least square fit when it is used to fit the parameter of a functional dependence to the measurements.

In general the distribution of the random variable y may not be a normal distribution. One can nevertheless show that the least square estimation is robust. However, it is biased. Depending on the nature of the distribution of the random variable y the parameters may be over- or underestimated. This is especially the case when working with histograms.

We have said that all measurements of the observable quantities y must be distributed according to a normal distribution so that the quantity $S(\mathbf{p})$ of equation 10.26 is distributed as a χ^2 random variable. In general this is often the case[6] when dealing with a large quantity of measurements. Thus, a least square fit is also called a χ^2 fit. In this case one can apply the χ^2-test described in section 10.3 to assess the goodness of the fitted function.

If $S(\mathbf{p})$ has a minimum respective to $\mathbf{p}$ then all partial derivatives of the function $S(\mathbf{p})$ respective to each of the components of the vector $\mathbf{p}$ are zero. Since the function is positive and quadratic in $\mathbf{p}$, it is clear that the function must have at least one minimum. Under this circumstances the minimum can be obtained by solving the following set of equations:

$$\frac{\partial}{\partial p_j} F(\mathbf{x}_i; p_1, \ldots, p_m) = 0 \quad \text{for } j = 1, \ldots, m \tag{10.27}$$

where m is the number of parameters, that is the dimension of the vector $\mathbf{p}$. When a solution is found, one should in principle verify that it is really a minimum. Solving equation 10.27 gives the following system of equations:

$$\sum_{i=1}^{N} \frac{y - F(\mathbf{x}, \mathbf{p})}{\sigma_i^2} \cdot \frac{\partial}{\partial p_j} S(p_1, \ldots, p_m) = 0 \quad \text{for } j = 1, \ldots, m \tag{10.28}$$

Once the system above has been solved, one can compute the value of $S(\mathbf{p})$ using equations 10.26 or, better, the value $S_{ML}(\mathbf{p})$ using equation 10.25. Computing the χ^2 confidence level of that value (*c.f.* section 10.3) using a χ^2 distribution with $N - m$ degrees of freedom gives the probability that the fit is acceptable.

10.6 Least square fit with linear dependence

If the function $F(\mathbf{x}, \mathbf{p})$ is a linear function of the vector $\mathbf{p}$, it can be written in the following form:

$$F(\mathbf{x}, \mathbf{p}) = \sum_{j=1}^{m} f_j(\mathbf{x}) \cdot p_j. \tag{10.29}$$

In that case, equation 10.28 become a system of linear equations of the form:

$$\mathbf{M} \cdot \mathbf{p} = \mathbf{c}, \tag{10.30}$$

where the coefficients of the matrix $\mathbf{M}$ are given by:

$$M_{jk} = \sum_{i=1}^{N} \frac{f_j(\mathbf{x}_i) f_k(\mathbf{x}_i)}{\sigma_i^2} \quad \text{for } j, k = 1, \ldots, m, \tag{10.31}$$

[6] This is a consequence of a theorem known as the law of large numbers.

and the components of the constant vector **c** are given by:

$$c_j = \sum_{i=1}^{N} \frac{y_i f_j(\mathbf{x}_i)}{\sigma_i^2} \quad \text{for } j = 1, \ldots, m. \tag{10.32}$$

Equation 10.30 is a system of linear equation which can be solved according to the algorithms exposed in sections 8.2 and 8.3. If one is interested only in the solution this is all there is to do.

A proper fit, however, should give an estimation of the error in estimating the parameters. The inverse of the matrix **M** is the error matrix for the fit. The error matrix is used to compute the estimation of variance on the function $F(\mathbf{x}, \mathbf{p})$ as follows:

$$\text{var}[F(\mathbf{x}, \mathbf{p})] = \sum_{j=1}^{m} \sum_{k=1}^{m} M_{jk}^{-1} f_j(\mathbf{x}) f_k(\mathbf{x}). \tag{10.33}$$

The estimated error on the function $F(\mathbf{x}, \mathbf{p})$ is the square root of the estimated variance.

The diagonal elements of the error matrix are the variance of the corresponding parameter. That is:

$$\text{var}(p_j) = M_{jj}^{-1} \quad \text{for } j = 1, \ldots, m. \tag{10.34}$$

The off diagonal elements describe the correlation between the errors on the parameters. One defines the correlation coefficient of parameter p_j and p_j by:

$$\text{cor}(p_j, p_k) = \frac{M_{jk}^{-1}}{\sqrt{M_{jj}^{-1} M_{kk}^{-1}}} \quad \text{for } j, k = 1, \ldots, m \text{ and } j \neq k. \tag{10.35}$$

All correlation coefficients are comprised between -1 and 1. If the absolute value of a correlation coefficient is close to 1, it means that one of the two corresponding two parameters is redundant for the fit. In other word, one parameter can be expressed as a function of the other.

10.7 Linear regression

A linear regression is a least square fit with a linear function of a single variable. The dimension of the vector **x** is one and the dimension of the vector **p** is two. The function to fit has only two parameters. The following convention is standard:

$$\begin{cases} p_1 & = & a, \\ p_2 & = & b, \\ F(\mathbf{x}, \mathbf{p}) & = & ax + b. \end{cases} \tag{10.36}$$

With these definitions, the system of equations 10.30 becomes:

$$\begin{cases} \sum_{i=1}^{N} \frac{x_i^2}{\sigma^2} a + \sum_{i=1}^{N} \frac{x_i}{\sigma^2} b &= \sum_{i=1}^{N} \frac{x_i y_i}{\sigma^2} \\ \sum_{i=1}^{N} \frac{x_i}{\sigma^2} a + \sum_{i=1}^{N} \frac{1}{\sigma^2} b &= \sum_{i=1}^{N} \frac{y_i}{\sigma^2}. \end{cases} \tag{10.37}$$

This system can easily be solved. Before giving the solution, let us introduce a short hand notation for the weighted sums:

$$\langle Q \rangle = \sum_{i=1}^{N} \frac{Q_i}{\sigma_i^2}. \tag{10.38}$$

Using this notation the solution of the system of equations 10.37 can be written as:

$$\begin{cases} a &= \frac{\langle xy \rangle \cdot \langle 1 \rangle - \langle x \rangle \cdot \langle y \rangle}{\langle xx \rangle \cdot \langle 1 \rangle - \langle x \rangle \cdot \langle x \rangle} \\ b &= \frac{\langle xx \rangle \cdot \langle y \rangle - \langle xy \rangle \cdot \langle x \rangle}{\langle xx \rangle \cdot \langle 1 \rangle - \langle x \rangle \cdot \langle x \rangle} \end{cases} \tag{10.39}$$

where the symmetry of the expression is quite obvious. It is interesting to note that if we had fitted x as a linear function of y, that is $x = \tilde{a}y + \tilde{b}$, we would have the following expression for the slope:

$$\tilde{a} = \frac{\langle xy \rangle \cdot \langle 1 \rangle - \langle x \rangle \cdot \langle y \rangle}{\langle yy \rangle \cdot \langle 1 \rangle - \langle y \rangle \cdot \langle y \rangle}. \tag{10.40}$$

If the dependence between x and y is truly a linear function, the product $a\tilde{a}$ ought to be 1. The square root of the product $a\tilde{a}$ is defined as the correlation coefficient of the linear regression, the sign of the square root being the sign of the slope. The correlation coefficient r is thus given by:

$$r = \frac{\langle xy \rangle \cdot \langle 1 \rangle - \langle x \rangle \cdot \langle y \rangle}{\sqrt{\left(\langle xx \rangle \cdot \langle 1 \rangle - \langle x \rangle \cdot \langle x \rangle\right)\left(\langle yy \rangle \cdot \langle 1 \rangle - \langle y \rangle \cdot \langle y \rangle\right)}}. \tag{10.41}$$

Since the least square fit is a biased estimator for the parameters, the square of the correlation coefficient is less than 1 in practice. The value of the correlation coefficient lies between -1 and 1. A good linear fit ought to have the absolute value of r close to 1.

Finally the error matrix of a linear regression is given by:

$$\mathbf{M}^{-1} = \frac{1}{\langle xx \rangle \cdot \langle 1 \rangle - \langle x \rangle \cdot \langle x \rangle} \begin{pmatrix} \langle xx \rangle & -\langle x \rangle \\ -\langle x \rangle & \langle 1 \rangle \end{pmatrix} \tag{10.42}$$

when the vector representing the parameters of the fit is defined as (b, a) in this order.

When fitting a functional dependence with one variable, x and many parameters, one can use a linear regression to reduce rounding errors when the observed values $y_1, \ldots, y_N$ cover a wide numerical range. Let a and b be the result of the linear regression of the values y_i as a function of x_i. One defines the new quantity $y'_i = y_i - (ax_i + b)$ for all i. The standard deviation of y'_i is the same as that of y_i since the subtracted expression is just a change of variable. In fact, the linear regression does not need to be a good fit at all. Then, the functional dependence can be fitted on the quantities $y'_1, \ldots, y'_N$. We shall give a detailed example on this method in section 10.8

Linear regression — General implementation

Figure 10-1 with the box **LinearRegression** grayed.

Linear regression is implemented within a single class using a similar implementation as that of the statistical moments. This means that individual measurements are accumulated and not stored. The drawback is that the object cannot compute the confidence level of the fit. This is not so much a problem since the correlation coefficient is usually sufficient to estimate the goodness of the fit.

The class has the following instance variables:

`sum1` is used to accumulate the sum of weights, that is, $\langle 1 \rangle$,

`sumX` is used to accumulate the weighted sum of x_i, that is, $\langle x \rangle$,

`sumY` is used to accumulate the weighted sum of y_i, that is, $\langle y \rangle$,

`sumXY` is used to accumulate the weighted sum of $x_i \times y_i$, that is, $\langle xy \rangle$,

`sumXX` is used to accumulate the weighted sum of x_i^2, that is, $\langle xx \rangle$,

`sumYY` is used to accumulate the weighted sum of y_i^2, that is, $\langle yy \rangle$,

`slope` the slope of the linear regression, that is, a,

`intercept` the value of the linear regression at $x = 0$, that is, b,

`tt correlationCoefficient` the correlation coefficient, that is, r in equation 10.41.

When either one of the instance variables `slope`, `intercept` or `correlationCoefficient` is needed, the method `computeResults` calculating the values of the three instance variables is called using lazy initialization. When new data is added to the object, these variables are reset. It is thus possible to investigate the effect of adding new measurements on the results.

The methods `asPolynomial` and `asEstimatedPolynomial` return an object used to compute the predicted value for any x. The estimated polynomial is using the error matrix of the least square fit to compute the error on the predicted value. Estimated polynomials are explained in section 10.8

Linear regression — Smalltalk implementation

Listing 10-17 shows the complete implementation in Smalltalk. The following code shows how to use the class PMLinearRegression to perform a linear regression over a series of measurements .

```
| linReg valueStream measurement slope intercept
  correlationCoefficient estimation value error|
linReg := PMLinearRegression new.
[ valueStream atEnd ]
       whileFalse: [ measurement := valueStream next.
                     linReg addPoint: measurement point
                             weight: measurement weight ].
slope := linReg slope.
intercept := linReg intercept.
correlationCoefficient := linReg correlationCoefficient.
estimation := linReg asEstimatedPolynomial.
value := estimation value: 0.5.
error := estimation error: 0.5.
```

This example assumes that the measurement of the random variable are obtained from a stream. The exact implementation of the stream is not shown here. The first line after the declaration creates a new instance of class DhbLinearRegressio Next comes the loop over all values found in the stream. This examples assumes that the values are stored on the stream as a single object implementing the following methods:

point returns a point containing the measurement, that is, the pair (x_i, y_i) for all i,

weight returns the weight of the measurement, that is, $1/\sigma_i^2$.

Each point is accumulated into the linear regression object with the method addPoi©nt:weight:.

After all measurements have been read, the results of the linear regression are fetched. The last three lines show how to obtain a polynomial object used to compute the value predicted by the linear regression at $x = 0.5$ and the error on that prediction.

The mechanism of lazy initialization is implemented by setting the three instance variables slope, intercept and correlationCoefficient to nil in the method reset.

Listing 10-17 Smalltalk implementation of linear regression

```
XXX
  % \input{Smalltalk/Estimation/DhbLinearRegression}
```

10.8 Least square fit with polynomials

In a polynomial fit the fit function is a polynomial of degree m . In this case, the parameters are usually numbered starting from 0; the number of free parameters is $m+1$ and the number of degrees of freedom is $N-m-1$. We have:

$$F\left(x; p_0, p_1, \ldots, p_m\right) = \sum_{k=0}^{n} p_k x^k. \tag{10.43}$$

The partial derivative of equation 10.27 is easily computed since a polynomial is a linear function of its coefficients:

$$\frac{\partial}{\partial p_j} F\left(\mathbf{x}_i; p_1, \ldots, p_m\right) = x_i^j \quad \text{for } j = 1, \ldots, m. \tag{10.44}$$

Such a matrix is called a Van Der Monde matrix. The system of equations 10.28 then becomes:

$$\sum_{k=0}^{m} p_k \cdot \sum_{i=1}^{N} \frac{x_i^{j+k}}{\sigma_i^2} = \sum_{i=1}^{N} \frac{x_i^j y_i}{\sigma_i^2}. \tag{10.45}$$

Equation 10.45 is of the same form as equation 10.30 where the coefficients of the matrix **M** are given by:

$$M_{jk} = \sum_{i=1}^{N} \frac{x_i^{j+k}}{\sigma_i^2}, \tag{10.46}$$

and the vector **c** has for components:

$$c_j = \sum_{i=1}^{N} \frac{x_i^j y_i}{\sigma_i^2}. \tag{10.47}$$

Polynomial least square fit provides a way to construct an ad-hoc representation of a functional dependence defined by a set of point. Depending on the type of data it can be more efficient than the interpolation methods discussed in chapter 3. In general the degree of the polynomial should be kept small to prevent large fluctuations between the data points.

Let us now shows a concrete example of polynomial fit.

In order to determine whether or not a fetus is developing itself normally within the womb, the dimension of the fetus' bones are measured during an ultrasound examination of the mother-to-be. The dimensions are compared against a set of standard data measured on a control population. Such data[7] are plotted in figure 10-18: the y-axis is the length of the femur expressed in mm; the x-axis represents the duration in weeks of the pregnancy based on the estimated date of conception. Each measurement has been obtained by

[7] These numbers are reproduced with permission of Prof. P.J. Steer. from the department of obstetrics and gynecology of the Chelsea & Westminster Hospital of London.

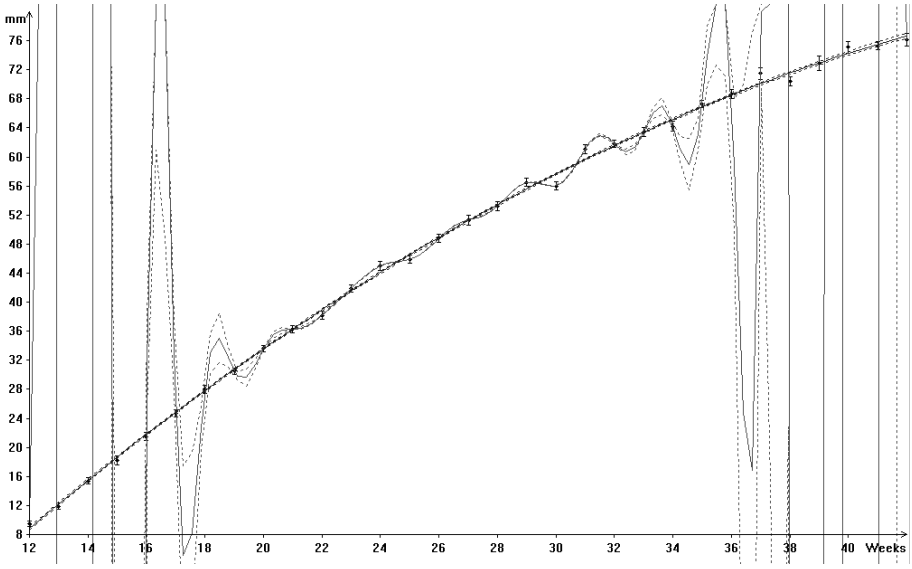


Figure 10-18 Example of polynomial fit

measuring the length of different fetuses at the same gestational age. The measurement are averaged and the error on the average is also calculated (*c.f.* section 9.1).

The obtained data do not follow a smooth curve since the data have been determined experimentally. The exact date of conception cannot be exactly determined so some fluctuation is expected, but the major limitation of the data is to obtain an sufficient number of measurements to smooth out the natural variations between individuals. The data from figure 10-18 have been fitted with a second order polynomial: the result is shown with a black thin curve. As one can see, the fit is excellent in spite of the fluctuation of the measurements. Figure 10-19 shows the fit results. This χ^2 confidence level is rather good. However, the correlation coefficients are quite high. This is usually the case with polynomial fits as each coefficient strongly depends on the other.

The thick gray line on figure 10-18 shows the interpolation polynomial[8] for comparison (interpolation is discussed in chapter 3). As the reader can see the interpolation polynomial gives unrealistic results because of the fluctuations of the experimental data.

Polynomial fits have something in common with interpolation: a fitted polynomial can seldom be used to extrapolate the data outside of the interval

[8] An attentive reader will notice that the interpolation curve dos not go through some data points. This is an artifact of the plotting over a finite sample of points which do not coincide with the measured data.

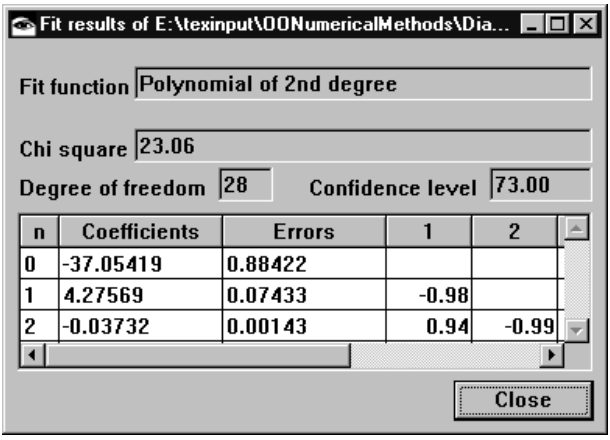


Figure 10-19 Fit results for the fit of figure 10-18

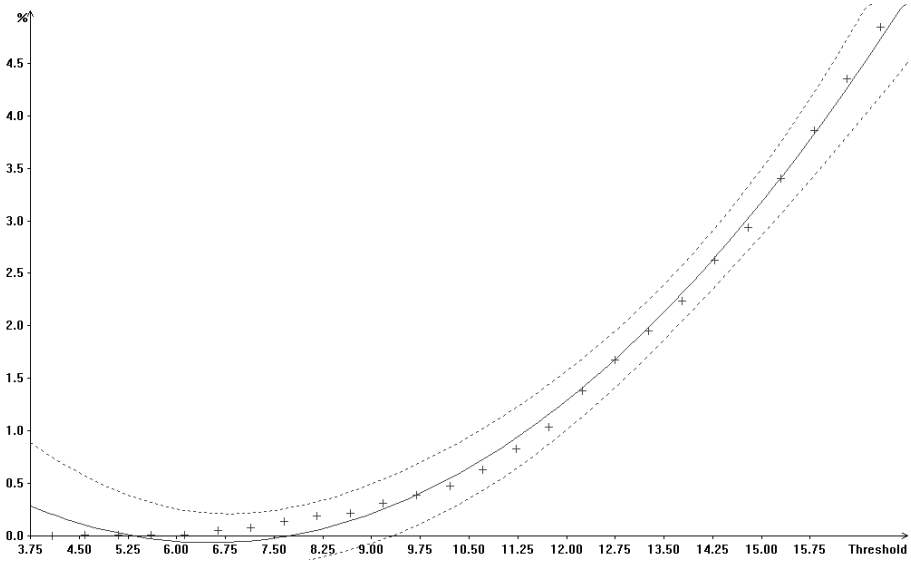


Figure 10-20 Limitation of polynomial fits

defined by the reference points. This is illustrated on figure 10-20 showing a second order polynomial fit made on a series of points. This series is discussed in section 3.1 (figure 3.1). The reader can see that the fitted polynomial does not reproduce the behavior of the data in the lower part of the region. Nevertheless, the data are well within the estimated error. Thus, the fit results are consistent. Unfortunately too many fit results are presented without their estimated error. This kind of information, however, is an essential part of a fit and should always be deliver along with the fitted function.

This is the idea behind what we call estimated polynomials. An estimated polynomial is a polynomial whose coefficients have been determined by a least square fit. An estimated polynomial keeps the error matrix of the fit is along with the coefficients of the fitted polynomial.

Polynomial least square fits — Smalltalk implementation

Figure 10-1 with the boxes **P**olynomial and **E**stimatedP grayed.

Listing **??** shows the complete implementation in Smalltalk. The following code example show how to perform the fit made in figure 10-18.

```
| fit valueStream dataHolder estimation value error|

<Accumulation of data into dataHolder>

fit := PMPolynomialLeastSquareFit new: 2.
dataHolder pointsAndErrorsDo: [ :each | fit add: each].
estimation := fit evaluate.
value := estimation value: 20.5.
error := estimation error: 20.5.
```

The data are accumulated into a object called dataHolder implementing the iterator method pointsAndErrorsDo. The argument of the block used by this method is an instance of class PMWeightedPoint described in section 10.3. The iterator method acts on all experimental data stored in dataHolder. Next, an instance of class PMPolynomialLeastSquareFit is created. The argument of the method is the degree of the polynomial; here a second order polynomial is used. After data have been accumulated into this object, the fit is performed by sending the method evaluate to the fit object. This method returns a polynomial including the error matrix. The last three lines compute the predicted femur length and its error in the middle of the 20 week of pregnancy.

The Smalltalk implementation assumes the points are stored in an object implementing the iterator method do:. Any instance of Collection of its subclasses will work. Each element of the collection must be an array containing the values x_i, y_i and $1/\sigma_i$. The class PMPolynomialLeastSquareFit keeps this collection in the instance variable pointCollection. A second instance variable, degreePlusOne, keeps the number of coefficients to be estimated by the fit.

The class creation method new: is used to create an instance by supplying the degree of the fit polynomial as argument. pointCollection is set to a new instance of an OrderedCollection. Then, new values can be added to the fit instance with the method add:.

The other class creation method, new:on: takes two arguments: the degree of the fit polynomial and the collection of points. The fit result can be fetched directly after the creation.

The method evaluate solves equation 10.28 by first computing the inverse of the matrix **M** to get the error matrix. The coefficients are then obtained from the multiplication of the constant vector by the error matrix.

Listing 10-21 m

```
alltalk implementation of a polynomial least square fit
    \label{ls:lspolynom}
\input{Smalltalk/Estimation/DhbPolynomialLeastSquareFit}
```

Listing **??** show the implementation of the class PMEstimatedPolynomial which is a subclass of the class PMPolynomial containing the error matrix of the fit performed to make a estimation of the polynomial's coefficients. The method error: returns the estimated error of its value based on the error matrix of the fit using equation 10.33. The convenience method valueAndError: returns an array containing the estimated value and its error in single method call. This is suitable for plotting the resulting curve.

Listing 10-22 m

```
alltalk implementation of a polynomial with error
    \label{ls:estpolynom}
\input{Smalltalk/Estimation/DhbEstimatedPolynomial}
```

10.9 Least square fit with non-linear dependence

In the case of a non-linear function, the fit can be reduced to a linear fit and a search by successive approximations.

Let us assume that we have an approximate estimation $\mathbf{p}_0$ of the parameters $\mathbf{p}$. Let us define the vector $\Delta\mathbf{p} = \mathbf{p} - \mathbf{p}_0$. One redefines the function $F(\mathbf{x}, \mathbf{p})$ as:

$$F(\mathbf{x}, \mathbf{p}) = F(\mathbf{x}, \mathbf{p}_0) + \left.\frac{\partial F(\mathbf{x}, \mathbf{p})}{\partial \mathbf{p}}\right|_{\mathbf{p}=\mathbf{p}_0} \cdot \Delta\mathbf{p}. \tag{10.48}$$

Equation 10.48 is a linear expansion[9] of the function $F(\mathbf{x}, \mathbf{p})$ around the vector $\mathbf{p}_0$ respective to the vector $\mathbf{p}$. In equation 10.48 $\left.\frac{\partial F(\mathbf{x}, \mathbf{p})}{\partial \mathbf{p}}\right|_{\mathbf{p}=\mathbf{p}_0}$ is the gradient of the function $F(\mathbf{x}, \mathbf{p})$relative to the vector $\mathbf{p}$ evaluated for $\mathbf{p} = \mathbf{p}_0$; this is a vector with the same dimension as the vector $\mathbf{p}$. Then, one minimizes the expression in equation 10.26 respective to the vector $\Delta\mathbf{p}$. This is of course a linear problem as described in section 10.6. Equation 10.30 becomes:

$$\mathbf{M} \cdot \Delta\mathbf{p} = \mathbf{c}, \tag{10.49}$$

[9] That is, the first couples of terms of a Taylor expansion of the function $F(\mathbf{x}, \mathbf{p})$ around the vector $\mathbf{p}_0$ in an m dimensional space.

where the components of the matrix **M** are now defined by:

$$M_{jk} = \sum_{i=1}^{N} \frac{1}{\sigma_i^2} \cdot \left.\frac{\partial F(\mathbf{x}_i, \mathbf{p})}{\partial p_j}\right|_{\mathbf{p}=\mathbf{p}_0} \cdot \left.\frac{\partial F(\mathbf{x}_i, \mathbf{p})}{\partial p_k}\right|_{\mathbf{p}=\mathbf{p}_0} \tag{10.50}$$

and the components of the vector **c** are defined by:

$$c_j = \sum_{i=1}^{N} \frac{y_i - F(\mathbf{x}_i, \mathbf{p}_0)}{\sigma_i^2} \cdot \left.\frac{\partial F(\mathbf{x}_i, \mathbf{p})}{\partial p_j}\right|_{\mathbf{p}=\mathbf{p}_0} . \tag{10.51}$$

The vector $\Delta\mathbf{p}$ is obtained by solving equation 10.49 using the algorithms described in sections 8.2 and 8.3. Then, we can use the vector $\mathbf{p}_0 + \Delta\mathbf{p}$ as the new estimate and repeat the whole process. One can show[10] that iterating this process converges toward the vector $\bar{\mathbf{p}}$ minimizing the function $S(\mathbf{p})$ introduced in equation 10.26.

As explained in section 10.6, the inverse of the matrix **M** is the error matrix containing the variance of each parameter and their correlation. The expression for the estimated variance on the function $F(\mathbf{x}, \mathbf{p})$ becomes:

$$\text{var}\left[F(\mathbf{x}, \mathbf{p})\right] = \sum_{j=1}^{m}\sum_{k=1}^{m} M_{jk}^{-1} \cdot \frac{\partial F(\mathbf{x}_i, \bar{\mathbf{p}})}{\partial p_j} \cdot \frac{\partial F(\mathbf{x}_i, \bar{\mathbf{p}})}{\partial p_k}. \tag{10.52}$$

A careful examination of the error matrix can tell whether or not the fit is meaningful.

Figure 10-23 shows an example of a least square fit performed on a histogram with a probability density function. The histogram of figure 10-23 was generated using a random generator distributed according to a Fisher-Tippett distribution (*c.f.* **??**) with parameters $\alpha = 0$ and $\beta = 1$. Only 1000 events have been accumulated into the histogram. The inset window in the upper right corner shows the fit results. The order of the parameter are α, β and the number of generated events. The solid curve laid onto the histogram is the prediction of the fitted function; the two dotted lines indicate the error on the prediction. The reader can verify that the fit is excellent. The number of needed iterations is quite low: the convergence of the algorithm is quite good in general.

Non-linear fit — General implementation

As we have seen the solution of a non-linear fit can be approximated by successive approximations. Thus, non-linear fits are implemented with a subclass of the iterative process class described in section 4.1. Data points must be kept in a structure maintained by the object implementing linear least

Figure 10-1 with the box **LeastSquare** grayed.

[10] A mathematically oriented reader can see that this is a generalization of the Newton zero-finding algorithm (*c.f.* section 5.3)to m dimensions .

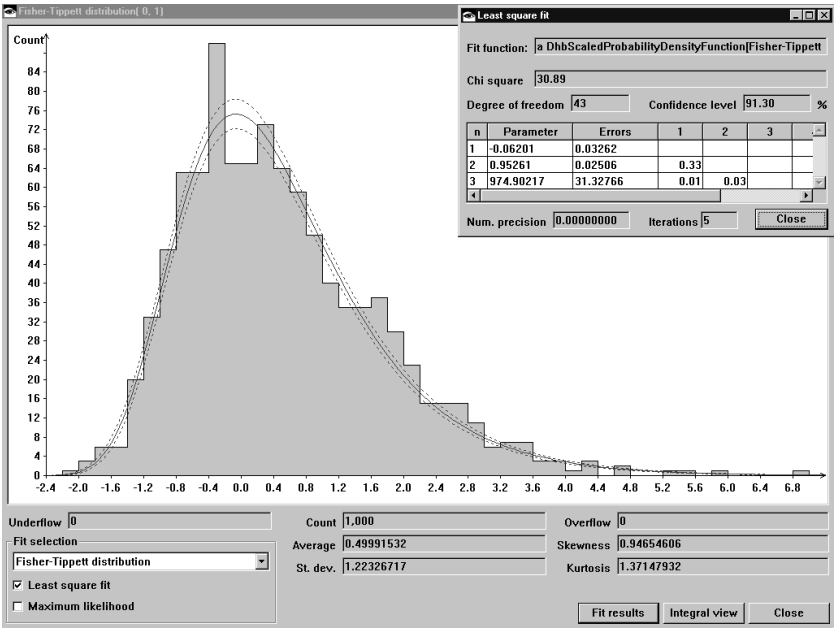


Figure 10-23 Example of a least square fit

square fit to be readily available at each iteration. Thus, the data point are kept in an instance variable.

The result of the iterative process are the parameters. Our implementation assumes that the supplied function contains and maintains its parameter. Thus, the instance variable corresponding to the result of the iterative process is the fit function itself. The parameters determined by the fit — the result proper — are kept within the object implementing the supplied function. In particular the determination of the initial values for the iterative process are the responsibility of the fit function. Thus, the method `initializeIterations` does not do anything.

In most cases, the number of parameters in a least square fits is relatively small. Thus, LUP decomposition — described in section 8.3 — is sufficient to solve equation 10.49 at each iteration. Except for the last iteration, there is no need to compute the error matrix (the inverse of the matrix **M**. The components of the error matrix can be obtained from the LUP decomposition when the algorithm converges.

Convergence is attained when the largest of the relative variation of the components of the vector becomes smaller than a given value. In addition to the fact that we are dealing with floating point numbers, the reason for using relative precision is that the components of the vector **p** usually have different ranges.

When the fit has been obetained, convenience methods allows to retrieve the sum of equation 10.25 (chiSquare) and the confidence level of the fit (confidenceLevel). Another convenience method, valueAndError computes the prediction of the fit and its estimated error using equation 10.52.

Non-linear fit — Smalltalk implementation

Listing **??** shows the complete implementation in Smalltalk. The following code example shows how the data of figure 10-23 were generated[11].

```
\label{exs:leastSquare}
 | genDistr hist fit |
 hist := PMHistogram new.
 hist freeExtent: true.
 genDistr := PMFisherTippettDistribution shape: 0 scale: 1.
 1000 timesRepeat: [ hist accumulate: genDistr random ].
 fit := PMLeastSquareFit histogram: hist
            distributionClass: DhbFisherTippettDistribution.
 fit evaluate.
```

The first two lines after the declaration define an instance of class PMHistogram with automatic adjustment of the limits (*c.f.* section 9.3). The next line defines an instance of a Fisher-Tippett distribution. Then, 1000 random numbers generated according to this distribution are accumulated into the histogram. Next, an instance if the class PMLeastSquareFit is defined with the histogram for the data points and the desired class of the probability density function. The corresponding scaled probability is created within the method (*c.f.* listing **??**). the final line performs the fit proper. After this line, the calling application can either use the fit object or extract the fit result to make predictions with the fitted distribution.

The class PMLeastSquareFit is a subclass of PMIterativeProcess described in section 4.1. It has the following instance variables:

dataHolder is the object containing the experimental data; this object must implement the iterator method pointsAndErrorsDo:; the block supplied to the iterator method takes as argument an instance of the class PMWeightedPoint described in section 10.3;

equations contains the components of the matrix **M**;

constants contains the components of the vector **c**;

errorMatrix contains the LUP decomposition of the matrix **M**;

chiSquare contains the sum of equation 10.25;

degreeOfFreedom contains the degree of freedom of the fit.

The instance variables errorMatrix, chiSquare and degreeOfFreedom are implemented using lazy initialization. The method finalizeIterations sets the

[11] . . .up to the plotting facilities. This could be the topic of a future book.d

instance variables equations and constants to nil to reclaim space at the end of the fit.

The supplied fit function — instance variable result — must implement the method valueAndGradient: which returns an array containing the value of the function at the supplied argument and the gradient vector. This is an optimization because the gradient can be computed frequently using intermediate results coming from the computation of the function's value.

The method valueAndError: is a good example of using the vector and matrix operations described in chapter 8.

Listing 10-24 m

```
alltalk implementation of a non-linear
least square fit \label{ls:lsfnonlin}
\input{Smalltalk/Estimation/DhbLeastSquareFit}
```

10.10 Maximum likelihood fit of a probability density function

In section 9.3 histograms have been discussed as a way to represent a probability density function directly from experimental data. In this section we shall show that the maximum likelihood estimation can easily be applied to the data gathered in a histogram in order to determine the parameters of a hypothesized probability density function.

In general the maximum likelihood fit of a probability density function to a histogram is much faster than the corresponding least square fit because the number of free parameters is lower, as we shall see in this section. In addition, the maximum likelihood estimation is unbiased and is therefore a better estimation than the least square fit estimation, especially when the histogram is sparsely populated. Thus, a maximum likelihood fit is the preferred way of finding the parameters of a probability density function from experimental data collected in a histogram.

Let m be the number of bins in the histogram and let n_i be the content of the i^{th} bin. Let $P_i(\mathbf{p})$ the probability of observing a value in the i^{th} bin. The likelihood function $L(\mathbf{p})$ is the probability of observing the particular histogram. Since the hypothesis of a probability density function does not constrain the total number of values collected into the histogram, the total number of collected values can be considered as constant. As a consequence, a maximum likelihood fit has one parameter less than a least square fit using the same function. Since the total number is unconstrained, the probability of observing the particular histogram is given by a multinomial probability. Thus, the likelihood function can be written as:

$$L(\mathbf{p}) = N! \prod_{i=1}^{m} \frac{P_i(\mathbf{p})^{n_i}}{n_i!}, \tag{10.53}$$

where $N = \sum_{i=1}^{m} n_i$ is the total number of values collected into the histogram. As we have seen in section 10.5, finding the maximum of $L(\mathbf{p})$ is equivalent of finding the maximum of the function $I(\mathbf{p})$. Since N is a constant, we use a renormalized function:

$$I(\mathbf{p}) = \ln \frac{M(\mathbf{p})}{N!} = \sum_{i=1}^{m} n_i \ln P_i(\mathbf{p}). \tag{10.54}$$

Finding the maximum of the function $I(\mathbf{p})$ is equivalent to solving the following system of non-linear equations:

$$\frac{\partial I(\mathbf{p})}{\partial \mathbf{p}} = \sum_{i=1}^{m} \frac{n_i}{P_i(\mathbf{p})} \cdot \frac{\partial P_i(\mathbf{p})}{\partial \mathbf{p}} = 0. \tag{10.55}$$

This system can be solved with a search by successive approximations, where a system of linear equations must be solved at each step. The technique used is similar to the one described in section 10.9. In this case, however, it is more convenient to expand the inverse of the probability density function around a previous approximation as follows:

$$\frac{1}{P_i(\mathbf{p})} = \frac{1}{P_i(\mathbf{p}_0)} - \frac{1}{P_i(\mathbf{p}_0)^2} \cdot \left.\frac{\partial P_i(\mathbf{p})}{\partial \mathbf{p}}\right|_{\mathbf{p}=\mathbf{p}_0} \cdot \Delta\mathbf{p}. \tag{10.56}$$

This expansion can only be defined over a range where the probability density function is not equal to zero. Therefore, this expansion of the maximum likelihood estimation cannot be used on a histogram where bins with non-zero count are located on a range where the probability density function is equal to zero[12]. Contrary to a least square fit, bins with zero count do not participate to the estimation.

Now equation 10.55 becomes a system of linear equations of the type:

$$\mathbf{M} \cdot \Delta\mathbf{p} = \mathbf{c}, \tag{10.57}$$

where the components of the matrix $\mathbf{M}$ are now defined by:

$$M_{jk} = \sum_{i=1}^{m} \frac{n_i}{P_i(\mathbf{p}_0)^2} \cdot \left.\frac{\partial P_i(\mathbf{p}_0)}{\partial p_j}\right|_{\mathbf{p}=\mathbf{p}_0} \cdot \left.\frac{\partial P_i(\mathbf{p}_0)}{\partial p_k}\right|_{\mathbf{p}=\mathbf{p}_0}, \tag{10.58}$$

and those of the vector $\mathbf{c}$ by:

$$c_j = \sum_{i=1}^{m} \frac{n_i}{P_i(\mathbf{p}_0)} \cdot \left.\frac{\partial P_i(\mathbf{p}_0)}{\partial p_j}\right|_{\mathbf{p}=\mathbf{p}_0}. \tag{10.59}$$

[12] Equation 10.54 shows that the bins over which the probability density function is zero give no information.

As discussed at the beginning of this section, the maximum likelihood estimation for a histogram cannot determine the total count in the histogram. The estimated total count, $\bar{N}$, is estimated with the following hypothesis:

$$n_i = \bar{N} P\left(\bar{\mathbf{p}}\right), \tag{10.60}$$

where $\bar{\mathbf{p}}$ is the maximum likelihood estimation of the distribution parameters. The estimation is performed using $\bar{N}$ as the only variable. The maximum likelihood estimation cannot be solved analytically, however, the least square estimation can.

As we have seen in section 10.4 the variance of the bin count is the estimated bin content. Thus, the function to minimize becomes:

$$S\left(\bar{N}\right) = \sum_{i=1}^{m} \frac{\left[n_i - \bar{N} P_i\left(\bar{\mathbf{p}}\right)\right]^2}{\bar{N} P_i\left(\bar{\mathbf{p}}\right)} \tag{10.61}$$

The value of $\bar{N}$ minimizing the expression of equation 10.61 is:

$$\bar{N} = \sqrt{\frac{\sum_{i=1}^{m} n_i^2 / P_i\left(\bar{\mathbf{p}}\right)}{\sum_{i=1}^{m} P_i\left(\bar{\mathbf{p}}\right)}}. \tag{10.62}$$

and the estimated error on $\bar{N}$ is given by

$$\sigma_{\bar{N}} = \sqrt{\frac{\sum_{i=1}^{m} n_i^2 / P_i\left(\bar{\mathbf{p}}\right)}{2\bar{N}}}. \tag{10.63}$$

After computing $\bar{N}$ using equation 10.62, the goodness of the maximum likelihood fit can be estimated by calculating the χ^2 confidence level of $S\left(\bar{N}\right)$ given by equation 10.61.

Figure 10-25 shows an example of a maximum likelihood fit performed on the same histogram as in figure 10-23. The inset window in the upper right corner shows the fit resultsin the same order as figure 10-23. The correlation coefficients, however, are not shown for the normalization since it is not determined as part of the fit. The solid curve laid onto the histogram is the prediction of the fitted function; the two dotted lines indicate the error on the prediction. The reader can see that the fit is as good as the least square fit. Of course, the χ^2 test is significantly higher with a correspondingly lower confidence level. This mostly comes from the fact that a maximum likelihood fit does not use the bins with zero count. In fact, the reader can see that the count in the histogram (normalization) estimated by the maximum likelihood fit is higher than in the case of the least square fit.

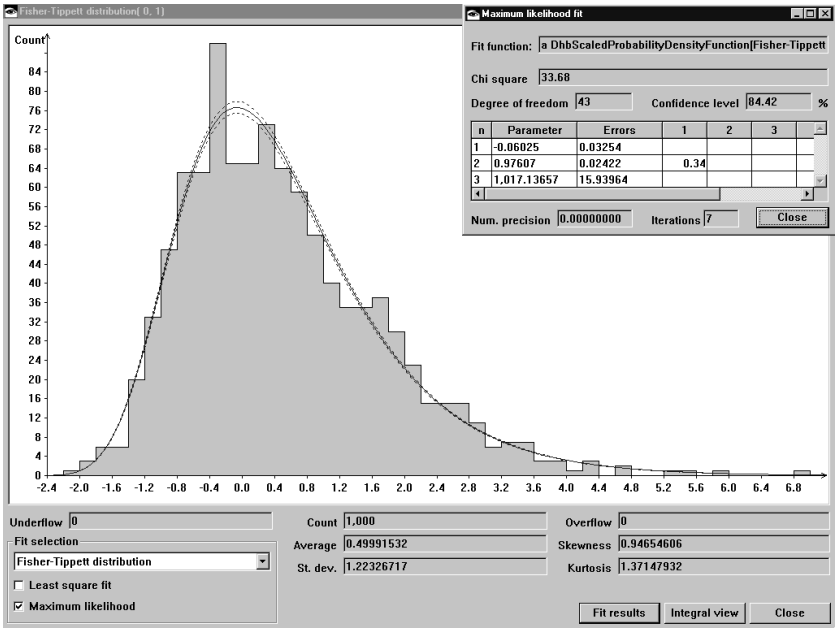


Figure 10-25 Example of a maximum likelihood fit

Maximum likelihood fit — General implementation

A maximum likelihood fit of a probability density function on a histogram is very similar to a least square fit of a histogram with a scaled probability distribution. There are two major differences: first the number of parameters is lower; second the computation of the matrix and vectors is not the same. Otherwise, most of the structure of a least square fit can be reused.

Instead of creating special methods to compute the gradient of the fitted function using a new set of parameters, our implementation uses the same gradient calculation than the one used by the least square fit. This is possible if the component of the gradient relative to the normalization is placed at the end. Since the computation of this component does not require additional calculation, the additional time required by the re-using of the gradient's computation is negligible. Since the fit function is a scaled probability distribution the current normalization is kept in an instance variable and the normalization of the fitted function is set to 1 for the duration of the iterations. When the algorithm is completed, the estimated normalization is put back into the fit function.

The computation of the normalization (equation 10.62) and that of its error (equation 10.63) is performed in the method `finalizeIterations`.

Figure 10-1 with the box MaximumL grayed.

Maximum likelihood fit — Smalltalk implementation

Listing **??** shows the complete implementation in Smalltalk. The following code example shows how figure 10-25 was generated up to the plotting facilities.

```
| genDistr hist fit |
hist := PMHistogram new.
hist freeExtent: true.
genDistr := PMFisherTippettDistribution shape: 0 scale: 1.
1000 timesRepeat: [ hist accumulate: genDistr random ].
fit := PMMaximumLikekihoodHistogramFit histogram: hist
          distributionClass: PMFisherTippettDistribution.
fit evaluate.
```

As the reader can see the only difference with code example **??** is the name of the class in the statement where the instance of the fit is created.

The class PMMaximumLikekihoodHistogramFit is a subclass of the class PMLeastSquareFit. It has the following additional instance variables:

count the estimated normalization, that is $\bar{N}$;

countVariance the estimated variance of $\bar{N}$.

The variance is kept instead of the error because the most frequent use of this quantity is in computing the estimated error on the predicted value. In the method valueAndError: this computation requires the combination of the error of the fit — that is, equation 10.33 — with the error on the normalization. An accessor method is provided for the variable count. The method normalizationError calculates the error on the normalization.

The method accumulate: uses the vector operations to calculate the terms of the sums in equations 10.58 and 10.59. Because of the lower number of parameters, the routine computeChanges: places in the vector $\Delta_{\mathbf{p}}$ an additional zero element corresponding to the normalization in the case of the least square fit.

The method finalizeIterations calculates the estimated value of the normalization (equation 10.61) and its variance (square of equation 10.62). After this, it sets the obtained normalization into the scaled probability distribution.

Listing 10-26 m

```
alltalk implementation of a maximum likelihood fit \label{ls:mlf}
\input{Smalltalk/Estimation/DhbMaximumLikekihoodHistogramFit}
```

CHAPTER 11

Optimization

Cours vite au but, mais gare à la chute.[1]
Alexandre Soljenitsyne

An optimization problem is a numerical problem where the solution is characterized by the largest or smallest value of a numerical function depending on several parameters. Such function is often called the goal function. Many kinds of problems can be expressed into optimization, that is, finding the maximum or the minimum of a goal function. This technique has been applied to a wide variety of fields going from operation research to game playing or artificial intelligence. In chapter 10 for example, the solution of maximum likelihood or least square fits was obtained by finding the maximum, respectively the minimum of a function.

In fact generations of high energy physicists have used the general purpose minimization program MINUIT[2] written by Fred James[3] of CERN to perform least square fits and maximum likelihood fits. To achieve generality, MINUIT uses several strategies to reach a minimum. In this chapter we shall discuss a few techniques and conclude with a program quite similar in spirit to MINUIT. Our version, however, will not have all the features offered by MINUIT.

If the goal function can be expressed with an analytical form, the problem of optimization can be reduced into calculating the derivatives of the goal function respective to all parameters, a tedious but manageable job. In most cases, however, the goal function cannot always be expressed analytically.

[1] Run fast to the goal, but beware of the fall.

[2] F.James, M. Roos, *MINUIT — a system for function minimization and analysis of the parameter errors and corrections*, Comput. Phys. Commun., 10 (1975) 343-367.

[3] I take this opportunity to thank Fred for the many useful discussions we have had on the subject of minimization.

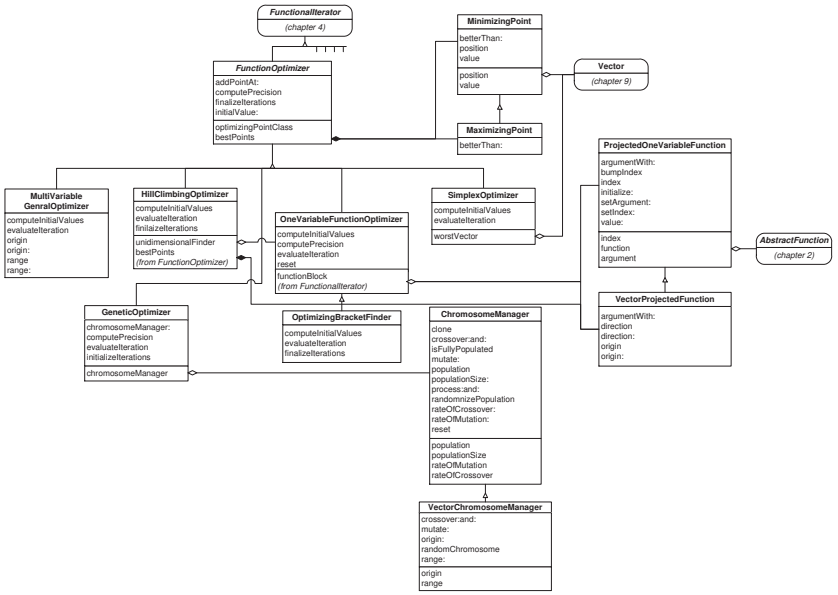


Figure 11-1 Smalltak classes used in optimization

Figure 11-1 shows the Smalltalk class diagram.

11.1 Introduction

Let us state the problem is general term. Let $f(\mathbf{x})$ be a function of a vector $\mathbf{x}$ of dimension n. The n-dimensional space is called the search space of the problem. Depending on the problem the space can be continuous or not. In this section we shall assume that the space is continuous.

If the function is derivable, the gradient of the function respective to the vector $\mathbf{x}$ must vanish at the optimum. Finding the optimum of the function can be replaced by the problem of finding the vector $\mathbf{x}$ such that:

$$\frac{df(\mathbf{x})}{d\mathbf{x}} = 0. \tag{11.1}$$

Unfortunately, the above equation is not a necessary condition for an optimum. It can be either a maximum. a minimum or a saddle point, that is a point where the function has a minimum in one projection and a maximum in another projection. Furthermore, the function may have several optima. Figure 11-2 shows an example of a function having two minima. Some problems require to find the absolute optimum of the function. Thus, one must verify that the solution of 11.1 corresponds indeed to an optimum with the expected properties. The reader can already see at this point that searching for an optimum in the general case is a very difficult task.

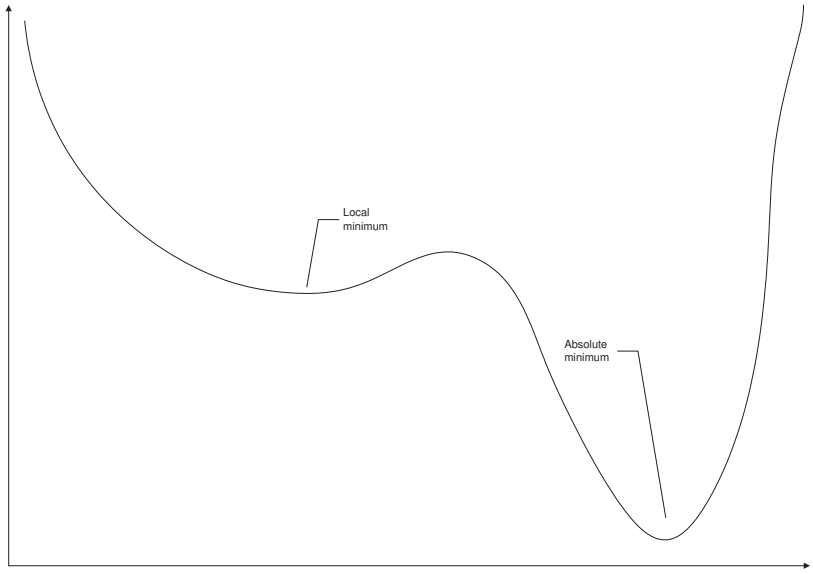


Figure 11-2 Local and absolute optima

All optimization algorithms can be classified in two broad categories:

- Greedy algorithms: these algorithms are characterized by a local search in the most promising direction. They are usually efficient and quite good at finding local optima. Among greedy algorithms, one must distinguish those requiring the evaluation of the function's derivatives.
- Random based algorithms: these algorithms are using a random approach. They are not efficient; however, they are good at finding absolute optima. Simulated annealing [**?**] and genetic algorithms[**?**] belong to this class.

The table 11-3 lists the properties of the algorithms presented in this chapter.

Table 11-3 Optimizing algorithms presented in this book

Name	**C**ategory	**D**erivatives
Extended Newton	greedy	yes
Powell's hill climbing	greedy	no
Simplex	greedy	no
Genetic algorithm	random based	no

11.2 Extended Newton algorithms

Extended Newton algorithms are using a generalized version of Newton's zero finding algorithm. These algorithms assume that the function is continuous and has only one optimum in the region where the search is initiated.

Let us expand the function $f(\mathbf{x})$ around a point $\mathbf{x}^{(0)}$ near the solution. We have in components:

$$f(\mathbf{x}) = f\left[\mathbf{x}^{(0)}\right] + \sum_j \left.\frac{\partial f(\mathbf{x})}{\partial x_j}\right|_{\mathbf{x}=\mathbf{x}^{(0)}} \left[x_j - x_j^{(0)}\right]. \tag{11.2}$$

Using the expansion above into equation 11.1 yields:

$$\sum_j \left.\frac{\partial^2 f(\mathbf{x})}{\partial x_i \partial x_j}\right|_{\mathbf{x}=\mathbf{x}^{(0)}} \left[x_j - x_j^{(0)}\right] + \left.\frac{\partial f(\mathbf{x})}{\partial x_i}\right|_{\mathbf{x}=\mathbf{x}^{(0)}} = 0. \tag{11.3}$$

This equation can be written as a system of linear equations of the form

$$\mathbf{M}\Delta = \mathbf{c}, \tag{11.4}$$

where $\Delta_j = x_j - x_j^{(0)}$. The components of the matrix $\mathbf{M}$ — called the Hessian matrix — are given by:

$$m_{ij} = \left.\frac{\partial^2 f(\mathbf{x})}{\partial x_i \partial x_j}\right|_{\mathbf{x}=\mathbf{x}^{(0)}}, \tag{11.5}$$

and the components of the vector $\mathbf{c}$ are given by:

$$c_i = -\left.\frac{\partial f(\mathbf{x})}{\partial x_i}\right|_{\mathbf{x}=\mathbf{x}^{(0)}}. \tag{11.6}$$

Like in section 10.9 one can iterate this process by replacing $\mathbf{x}^{(0)}$ with $\mathbf{x}^{(0)} + \Delta$. This process is actually equivalent to the Newton zero finding method (*c.f.* section 5.3). The final solution is a minimum if the matrix $\mathbf{M}$ is positive definite; else it is a maximum.

This technique is used by MINUIT in the vicinity of the goal function's optimum. It is the region where the algorithm described above works well. Far from the optimum, the risk of reaching a point where the matrix $\mathbf{M}$ cannot be inverted is quite high in general. In addition, the extended Newton algorithm requires that the second order derivatives of the function can be computed analytically; at least the first order derivatives must be provided, otherwise the cost of computation at each step becomes prohibitive. A concrete implementation of the technique is not given here. The reader can find in this book all the necessary tools to make such an implementation. It is left as a exercise for the reader. In the rest of this chapter, we shall present other methods which work without an analytical knowledge of the function.

11.3 Hill climbing algorithms

Hill climbing is a generic term covering many algorithms trying to reach an optimum by determining the optimum along successive directions. The general algorithm is outlined below.

1. select an initial point $\mathbf{x}_0$ and a direction $\mathbf{v}$;
2. find $\mathbf{x}_1$, the optimum of the function along the selected direction;
3. if convergence is attained, terminate the algorithm;
4. set $\mathbf{x}_0 = \mathbf{x}_1$, select a different direction and go back to step 2.

The simplest of these algorithms simply follows each axis in turn until a convergence is reached. More elaborate algorithms exist[**?**]. One of them is described in section 11.6.

Hill climbing algorithms can be applied to any continuous function, especially when the function's derivatives are not easily calculated. The core of the hill climbing algorithm is finding the optimum along one direction. Let $\mathbf{v}$ be the direction, then finding the optimum of the vector function $f(\mathbf{x})$ along the direction $\mathbf{v}$ starting from point $\mathbf{x}_0$ is equivalent to finding the optimum of the one-variable function $g(\lambda) = f(\mathbf{x}_0 + \lambda\mathbf{v})$.

Therefore, in order to implement a hill climbing algorithm, we first need to implement an algorithm able to find the optimum of a one-variable function. This is the topic of the sections 11.4 and 11.5. Before this, we need to discuss the implementation details providing a common framework to all classes discussed in the rest of this chapter.

Optimizing — General implementation

At this point the reader may be a little puzzled by the use of optimum instead of speaking of minimum or maximum. We shall now disclose a general implementation which works both for finding a minimum or a maximum. This should not come to a surprise since, in mathematics, a minimum or a maximum are both very similar — position where the derivative of a function vanishes — and can be easily turned into each other — e.g. by negating the function.

To implement a general purpose optimizing framework, we introduce two new classes: MinimizingPoint and MaximizingPoint, a subclass of MinimizingPoint. These two classes are used as Strategy by the optimizing algorithms. The class MinimizingPoint has two instance variables

value the value of the goal function, that is $g(\lambda)$ or $f(\mathbf{x})$;

position the position at which the function has been evaluated, that is λ or $\mathbf{x}$.

The class MinimizingPoint contains most of the methods. The only method overloaded by the class MaximizingPoint is the method betterThan, which

tells whether an optimizing point is better than another. The method betterThan can be used in all parts of the optimizing algorithms to find out which point is the optimum so far. In algorithms working in multiple dimensions, the method betterThan is also used to sort the points from the best to the worst.

A convenience instance creation method allows to create instances for a given function with a given argument. The instance is then initialized with the function's value evaluated at the argument. Thus, all optimizing algorithms described here do not call the goal function explicitly.

Otherwise the implementation of the one dimensional optimum search uses the general framework of the iterative process. More specifically it uses the class FunctionalIterator described in section 4.2.

A final remark concerns the method initializeIteration. The golden search algorithm assume that the 3 points λ_0, λ_1 and λ_2 have been determined. What if they have not been? In this case, the method initializeIteration uses the optimum bracket finder described in section 11.5

Common optimizing classes

In Pharo we have two classes of optimizing points: PMMinimizingPoint and its subclass PMMaximizingPoint. These classes are shown in listing 11-4. The class PMFunctionOptimizer is in charge of handling the management of the optimizing points. This clas is shown in listing 11-5.

The class PMMinimizingPoint has the following instance variables:

position contains the position at which the function is evaluated; this instance variable is a number if the function to optimize is a one variable function and an array or a vector if the function to evaluate is a function of many variables;

value contains the value of the function evaluated at the point's position;

Accessor methods corresponding to these variables are supplied. As we noted in section 11.3, the only method redefined by the subclass PMMaximizingPoint is the method betterThan: used to decide whether a point is better than another.

Optimizing points are created with the convenience method vector:function: which evaluates the function supplied as second argument at the position supplied as the first argument.

Listing 11-4 Smalltalk classes common to all optimizing classes

```
Object subclass: #PMMinimizingPoint
   instanceVariableNames: 'value position'
   classVariableNames: ''
   package: 'Math-Numerical-Math-FunctionIterator'
```

```
PMMinimizingPoint class >> new: aVector value: aNumber
   ^ self new vector: aVector; value: aNumber; yourself
```

```
PMMinimizingPoint class >> vector: aVector function: aFunction
   ^ self new: aVector value: (aFunction value: aVector)
```

```
PMMinimizingPoint >> betterThan: anOptimizingPoint
   ^ value < anOptimizingPoint value
```

```
PMMinimizingPoint >> position
   ^ position
```

```
PMMinimizingPoint >> printOn: aStream
   position printOn: aStream.
   aStream
      nextPut: $:;
      space.
      value printOn: aStream
```

```
PMMinimizingPoint >> value
   ^ value
```

```
PMMinimizingPoint >> value: aNumber
   value := aNumber.
```

```
PMMinimizingPoint >> vector: aVector
    position := aVector
```

```
PMMinimizingPoint subclass: #PMMaximizingPoint
   instanceVariableNames: ''
   classVariableNames: ''
   package: 'Math-Numerical-Math-FunctionIterator'
```

```
PMMaximizingPoint >> betterThan: anOptimizingPoint
   ^ value > anOptimizingPoint value
```

The class PMFunctionOptimizer is in charge of handling the optimizing points. it has the following instance variables:

optimizingPointClass is the class of the optimizing points used as Strategy by the optimizer; it is used to create instances of points at a given position for a given function;

bestPoints contains a sorted collection of optimizing points; the best point is the first and the worst point is the last; all optimizers rely on the fact that sorting is done by this sorted collection.

The method addPointAt: creates an optimizing point at the position supplied as argument and adds this point to the collection of best points. Since that collection is sorted, one is always certain to find the best result in the first position. This fact is used by the method finalizeIterations, which retrieves the result from the collection of best points.

Instances of the function optimizer are created with the two convenience methods minimizingFuntion: and maximizingFuntion: helping to define the

type of optimum. An additional convenience method, `forOptimizer:` allows to create a new optimizer with the same strategy — that is, the same class of optimizing points — and the same function as the optimizer supplied as argument. This method is used to create optimizers used in intermediate steps.

Because finding an optimum cannot be determined numerically with high precision [**?**] the class PMFunctionOptimizer redefines the method defaultPrecision to be 100 times the default numerical precision.

Listing 11-5 Smalltalk abstract class for all optimizing classes

```
PMFunctionalIterator subclass: #PMFunctionOptimizer
   instanceVariableNames: 'optimizingPointClass bestPoints'
   classVariableNames: ''
   package: 'Math-Numerical-Math-FunctionIterator'
```

```
PMFunctionOptimizer class>>defaultPrecision}
   ^ super defaultPrecision * 100
```

```
PMFunctionOptimizer class>>forOptimizer: aFunctionOptimizer
   ^ self new initializeForOptimizer: aFunctionOptimizer
```

```
PMFunctionOptimizer class>>maximizingFunction: aFunction
   ^ super new initializeAsMaximizer; setFunction: aFunction
```

```
PMFunctionOptimizer class>>minimizingFunction: aFunction
   ^ super new initializeAsMinimizer; setFunction: aFunction
```

```
PMFunctionOptimizer>>addPointAt: aNumber

    bestPoints add: (optimizingPointClass vector: aNumber
                                        function: functionBlock)
```

```
PMFunctionOptimizer>>bestPoints
   ^ bestPoints
```

```
PMFunctionOptimizer>>finalizeIterations
   result := bestPoints first position.
```

```
PMFunctionOptimizer>>functionBlock
   ^ functionBlock
```

```
PMFunctionOptimizer>>initialize
    bestPoints := SortedCollection sortBlock:
                                      [ :a :b | a betterThan: b ].
    ^ super initialize
```

```
PMFunctionOptimizer>>initializeAsMaximizer
    optimizingPointClass := PMMaximizingPoint.
    ^ self initialize
```

```
PMFunctionOptimizer>>initializeAsMinimizer
   optimizingPointClass := PMMinimizingPoint
```

```
PMFunctionOptimizer>>initializeForOptimizer: aFunctionOptimizer
   optimizingPointClass := aFunctionOptimizer pointClass.
   functionBlock := aFunctionOptimizer functionBlock.
   ^ self initialize
```

```
PMFunctionOptimizer>>initialValue: aVector
    result := aVector copy.
```

```
PMFunctionOptimizer>>pointClass
    ^ optimizingPointClass
```

```
PMFunctionOptimizer>>printOn: aStream
    super printOn: aStream.
    bestPoints do: [ :each | aStream cr. each printOn: aStream ].
```

In order to find an optimum along a given direction, one needs to construct an object able to transform a vector function into a one variable function. The class PMProjectedOneVariableFunction and its subclass PMVectorProjectedFunction provide this functionality. They are shown in listing 11-6. The class PMProjectedOneVariabl has the following instance variables:

function the goal function $f(\mathbf{x})$;

argument the vector argument of the goal function, that is the vector $\mathbf{x}$;

index the index of the axis along which the function is projected.

The instance variables argument and index can be read and modified using direct accessor methods. The goal function is set only at creation time: the instance creation method function: take the goal function as argument. A convenience method bumpIndex allows to alter the index in circular fashion[4].

The class PMVectorProjectedFunction has no additional variables. Instead it is reusing the instance variable index as the direction along which the function is evaluated. For clarity, the accessor methods have been renamed direction, direction:, origin and origin:.

For both classes, the method argumentAt: returns the argument vector for the goal function, that is the vector $\mathbf{x}$. The method value: returns the value of the function $g(\lambda)$ for the supplied argument λ.

Listing 11-6 Smalltalk projected function classes

```
Object subclass: #PMProjectedOneVariableFunction
   instanceVariableNames: 'index function argument'
   classVariableNames: ''
   package: 'Math-Numerical'
```

```
PMProjectdOneVariableFunction class>>function: aVectorFunction
    ^ super new initialize: aVectorFunction
```

[4] We do not give the implementation of the simplest of the hill climbing algorithms using alternatively each axes of the reference system. This implementation, which uses the method bumpIndex, is left as an exercise for the reader.

```
PMProjectdOneVariableFunction >> argumentWith: aNumber
    ^ argument at: index put: aNumber; yourself
```

```
PMProjectdOneVariableFunction >> bumpIndex
    index isNil
        ifTrue: [ index := 1]
        ifFalse: [ index := index + 1.
                   index > argument size
                     ifTrue: [ index := 1].
                 ]
```

```
PMProjectdOneVariableFunction >> index
    index isNil
        ifTrue: [ index := 1].
    ^ index
```

```
PMProjectdOneVariableFunction >> initialize: aFunction
    function := aFunction.
    ^ self
```

```
PMProjectdOneVariableFunction >> setArgument: anArrayOrVector
    argument := anArrayOrVector copy
```

```
PMProjectdOneVariableFunction >> setIndex: anInteger
    index := anInteger
```

```
PMProjectdOneVariableFunction >> value: aNumber
    ^ function value: ( self argumentWith: aNumber)
```

```
PMProjectedOneVariableFunction subclass: #PMVectorProjectedFunction
   instanceVariableNames: ''
   classVariableNames: ''
   package: 'Math-Numerical'
```

```
PMVectorProjectedFunction >> argumentWith: aNumber
    ^ aNumber * self direction + self origin
```

```
PMVectorProjectedFunction >> direction
    ^ index
```

```
PMVectorProjectedFunction >> direction: aVector
    index := aVector.
```

```
PMVectorProjectedFunction >> origin}
    ^ argument
```

```
PMVectorProjectedFunction >> origin: aVector
    argument := aVector
```

```
PMVectorProjectedFunction >> printOn:aStream
    self origin printOn: aStream.
    aStream nextPutAll: ' ('.
    self direction printOn: aStream.
    aStream nextPut: $).
```

11.4 Optimizing in one dimension

To find the optimum of a one-variable function, $g(\lambda)$, whose derivative is unknown, the most robust algorithm is an algorithm similar to the bisection algorithm described in section 5.2.

Let us assume that we have found three points λ_0, λ_1 and λ_2 such that $\lambda_0 < \lambda_1 < \lambda_2$ and such that $g(\lambda_1)$ is better than both $g(\lambda_0)$ and $g(\lambda_2)$. If the function g is continuous over the interval $[\lambda_0, \lambda_2]$, then we are certain that an optimum of the function is located in the interval $[\lambda_0, \lambda_2]$. As for the bisection algorithm, we shall try to find a new triplet of values with similar properties while reducing the size of the interval. A point is picked in the largest of the two intervals $[\lambda_0, \lambda_1]$ or $[\lambda_1, \lambda_2]$ and is used to reduce the initial interval.

If $\lambda_1 - \lambda_0 \leq \lambda_2 - \lambda_1$ we compute $\lambda_4 = \lambda_1 + \omega(\lambda_2 - \lambda_1)$ where ω is the golden section[5] from Pythagorean lore. Choosing ω instead of $1/2$ ensures that successive intervals have the same relative scale. A complete derivation of this argument can be found in [**?**]. If λ_4 yields a better function value than λ_1, the new triplet of point becomes λ_1, λ_4 and λ_2; otherwise, the triplet becomes λ_0, λ_1 and λ_4.

If we have $\lambda_1 - \lambda_0 > \lambda_2 - \lambda_1$ we compute $\lambda_4 = \lambda_1 + \omega(\lambda_0 - \lambda_1)$. Then the new triplets can be either λ_0, λ_4 and λ_1, or λ_4, λ_1 and λ_2.

The reader can verify that the interval decreases steadily although not as fast as in the case of bisection where the interval is halved at each iteration. Since the algorithm is using the golden section it is called golden section search.

By construction the golden section search algorithm makes sure that the optimum is always located between the points λ_0 and λ_2. Thus, at each iteration, the quantity $\lambda_2 - \lambda_0$ give an estimate of the error on the position of the optimum.

Optimizing in one dimension

Listing 11-7 shows the class PMOneVariableFunctionOptimizer implementing the search for an optimum of a one-variable function using the golden section search. The following code example shows how to use this class to find the maximum of the gamma distribution discussed in section 9.7.

```
| distr finder maximum |
   distr := PMGammaDistribution shape: 2 scale: 5.
   finder := PMOneVariableFunctionOptimizer maximizingFunction:
    distr.
   maximum := finder evaluate
```

The first line after the declarations creates a new instance of a gamma distribution with parameters $\alpha = 2$ and $\beta = 5$. The next line creates an instance of

[5] $\omega = \frac{3-\sqrt{5}}{2} \approx 0.38197$

the optimum finder. The selector used to create the instance selects a search for a maximum. The last line is the familiar statement to evaluate the iterations — that is, performing the search for the maximum — and to retrieve the result. Since no initial value was supplied the search begins at a random location.

The class PMOneVariableFunctionOptimizer is a subclass of the class PMFunctionOptimizer. It does not need any additional instance variables. The golden section is kept as a class variable and is calculated at the first time it is needed.

At each iteration the method nextXValue is used to compute the next position at which the function is evaluated. This corresponding new optimizing point is added to the collection of best points. Then, the method indexOfOuterPoint is used to determine which point must be discarded: it is always the second point on either side of the best point. The precision of the result is estimated from the bracketing interval in the method computePrecision, using relative precision (of course!).

The method addPoint: of the superclass can be used to supply an initial bracketing interval. The method computeInitialValues first checks whether a valid bracketing interval has been supplied into the collection of best points. If this is not the case, a search for a bracketing interval is conducted using the class PMOptimizingBracketFinder described in section **??**. The instance of the bracket finder is created with the method forOptimizer: so that its strategy and goal function are taken over from the golden section optimum finder.

Listing 11-7 Smalltalk golden section optimum finder

```
PMFunctionOptimizer subclass: #PMOneVariableFunctionOptimizer
   instanceVariableNames: ''
   classVariableNames: 'GoldenSection'
   package: 'Math-Numerical-Math-FunctionIterator'
```

```
PMOneVariableFunctionOptimizer class >> defaultPrecision
    ^ PMFloatingPointMachine new defaultNumericalPrecision * 10
```

```
PMOneVariableFunctionOptimizer class >> goldenSection
    GoldenSection isNil ifTrue: [GoldenSection := (3 - 5 sqrt) / 2].
    ^ GoldenSection
```

```
PMOneVariableFunctionOptimizer >> computeInitialValues
    [ bestPoints size  > 3 ] whileTrue: [ bestPoints removeLast ].
    bestPoints size = 3
        ifTrue: [ self hasBracketingPoints
                    ifFalse: [ bestPoints removeLast ].
                ].
    bestPoints size < 3
        ifTrue: [ (PMOptimizingBracketFinder forOptimizer: self)
    evaluate ].
```

```
PMOneVariableFunctionOptimizer>>computePrecision
    ^ self precisionOf: ((bestPoints at: 2) position - (bestPoints
                                                  at: 3) position)
    abs
           relativeTo: (bestPoints at: 1) position abs
```

```
PMOneVariableFunctionOptimizer>>evaluateIteration
    self addPointAt: self nextXValue.
    bestPoints removeAtIndex: self indexOfOuterPoint.
    ^ self computePrecision
```

```
PMOneVariableFunctionOptimizer>>hasBracketingPoints
    | x1 |
    x1 := (bestPoints at: 1) position.
    ^ ((bestPoints at: 2) position - x1) * ((bestPoints at: 3)
                                                  position - x1) <
    0
```

```
PMOneVariableFunctionOptimizer>>indexOfOuterPoint
    | inferior superior x |
    inferior := false.
    superior := false.
    x := bestPoints first position.
    2 to: 4 do:
        [ :n |
          (bestPoints at: n) position < x
                ifTrue: [ inferior
                            ifTrue: [ ^ n ].
                          inferior := true.
                        ]
                ifFalse:[ superior
                            ifTrue: [ ^ n ].
                          superior := true.
                        ].
        ]
```

```
PMOneVariableFunctionOptimizer>>nextXValue
    | d3 d2 x1 |
    x1 := (bestPoints at: 1) position.
    d2 := (bestPoints at: 2) position - x1.
    d3 := (bestPoints at: 3) position - x1.
    ^ (d3 abs > d2 abs ifTrue: [ d3 ]
                       ifFalse:[ d2 ]) * self class goldenSection +
    x1
```

```
PMOneVariableFunctionOptimizer>>reset
    [ bestPoints isEmpty ] whileFalse: [ bestPoints removeLast ].
```

11.5 Bracketing the optimum in one dimension

As we have seen in section 11.4 the golden section algorithm requires the knowledge of a bracketing interval. This section describes a very simple algorithm to obtain a bracketing interval with certainty if the function is continuous and does indeed have an optimum of the sought type.

The algorithm goes as follows. Take two points λ_0 and λ_1. If they do not exist, pick up some random values (random generators are described in section 9.4). Let us assume that $g(\lambda_1)$ is better than $g(\lambda_0)$.

1. Let $\lambda_2 = 3\lambda_1 - 2\lambda_0$, that is, λ_2 is twice as far from λ_1 than λ_0 and is located on the other side, toward the optimum.
2. If $g(\lambda_1)$ is better than $g(\lambda_2)$ we have a bracketing interval; the algorithm is stopped.
3. Otherwise, set $\lambda_0 = \lambda_1$ and $\lambda_1 = \lambda_2$ and go back to step 1.

The reader can see that the interval $[\lambda_0, \lambda_1]$ is increasing at each step. Thus, if the function has no optimum of the sought type, the algorithm will cause a floating overflow exception quite rapidly.

The implementation in each language have too little in common. The common section is therefore omitted.

Bracketing the optimum

Listing **??** shows the Smalltalk code of the class implementing the search algorithm for an optimizing bracket. The class PMOptimizingBracketFinder is a subclass of class PMOneVariableFunctionOptimizer from section 11-7. This was a convenient, but not necessary, choice to be able to reuse most of the management and accessor methods. The methods pertaining to the algorithm are of course quite different.

Example of use of the optimizing bracket finder can be found in method computeInitialValues of class PMOneVariableFunctionOptimizer (*c.f.* listing 11-7).

The method setInitialPoints: allows to use the collection of best points of another optimizer inside the class. This breach to the rule of hiding the implementation can be tolerated here because the class PMOptimizingBracketFinder is used exclusively with the class PMOneVariableFunctionOptimizer. It allows the two class to use the same sorted collection of optimizing points. If no initial point has been supplied, it is obtained from a random generator.

The precision calculated in the method evaluateIteration is a large number, which becomes negative as soon as the condition to terminate the algorithm is met. Having a negative precision causes an iterative process as defined in chapter 4 to stop.

Listing 11-8 m

```
alltalk optimum bracket finder \label{ls:optimizerbracket}
\input{Smalltalk/Minimization/DhbOptimizingBracketFinder}
```

11.6 Powell's algorithm

Powell's algorithm is one of many hill climbing algorithms [**?**]. The idea underlying Powell's algorithm is that once an optimum has been found in one direction, the chance for the biggest improvement lies in the direction perpendicular to that direction. A mathematical formulation of this sentence can be found in [**?**] and references therein. Powell's algorithm provides a way to keep track of the next best direction at each iteration step.

The original steps of Powell's algorithm are as follow:

1. Let $\mathbf{x}_0$ the best point so far and initialize a series of vectors $\mathbf{v}_1, \ldots, \mathbf{v}_n$ forming the system of reference (n is the dimension of the vector $\mathbf{x}_0$); in other words the components of the vector $\mathbf{v}_k$ are all zero except for the k components, which is one.
2. Set $k = 1$.
3. Find the optimum of the goal function along the direction $\mathbf{v}_k$ starting from point $\mathbf{x}_{k-1}$. Let $\mathbf{x}_k$ be the position of that optimum.
4. Set $k = k + 1$. If $k \leq n$, go back to step 3.
5. For $k = 1, \ldots, n - 1$, set $\mathbf{v}_k = \mathbf{v}_{k-1}$.
6. Set $\mathbf{v}_n = \frac{\mathbf{x}_n - \mathbf{x}_0}{|\mathbf{x}_n - \mathbf{x}_0|}$.
7. Find the optimum of the goal function along the direction $\mathbf{v}_n$. Let $\mathbf{x}_{n+1}$ be the position of that optimum.
8. If $|\mathbf{x}_n - \mathbf{x}_0|$ is less than the desired precision, terminate.
9. Otherwise, set $\mathbf{x}_0 = \mathbf{x}_{n+1}$ and go back to step 1.

There is actually two hill climbing algorithms within each other. The progression obtained by the inner loop is taken as the direction in which to continue the search.

Powell recommends to use this algorithm on goal functions having a quadratic behaviour near the optimum. It is clear that this algorithm cannot be used safely on any function. If the goal function has narrow valleys, all directions $\mathbf{v}_1, \ldots, \mathbf{v}_n$ will become colinear when the algorithm ends up in such a valley. Thus, the search is likely to end up in a position where no optimum is located. Press et al. [**?**] mention two methods avoiding such problems: one method is quite complex and the other slows down the convergence of the algorithm.

In spite of this caveat, we implement Powell's algorithm in its original form. However, we recommend its use only in the vicinity of the minimum. In section 11.9 we show how other techniques can be utilized to read the vicinity of

the optimum, where Powell's algorithm can safely be used to make the final determination of the optimum's position.

Powell's algorithm — General implementation

Since the class implementing the vector projected function $g(\lambda)$ described in section **??** keep the vector $\mathbf{x}_0$ and $\mathbf{v}$ in instance variables, there is no need to allocate explicit storage for the vectors $\mathbf{x}_1, \ldots, \mathbf{x}_n$ and $\mathbf{v}_1, \ldots, \mathbf{v}_n$. Instead, the class implementing Powell's algorithm keep an array of vector projected functions with the corresponding parameters. Then, the manipulation of the vector $\mathbf{x}_1, \ldots, \mathbf{x}_n$ and $\mathbf{v}_1, \ldots, \mathbf{v}_n$ is made directly on the projected function. Since the origin of the projected function is always the starting value, $\mathbf{x}_k$, the initial value for the search of the optimum of the function $g(\lambda)$ is always 0.

The method `initializeIterations` allocated a series of vector projected functions starting with the axes of the reference system. This method also creates an instance of a one dimensional optimum finder kept in the instance variable, unidimensionalFinder. The goal function of the finder is alternatively assigned to each of the projected functions.

We made a slight modification to Powell's algorithm. If the norm of the vector $\mathbf{x}_n - \mathbf{x}_0$ at step 6 is smaller than the desired precision, the directions are only rotated, the assignment of step 6 is not done and the search of step 7 is omitted.

The precision computed at the end of each iterations is the maximum of the relative change on all components between the vectors $\mathbf{x}_n$ and $\mathbf{x}_0$.

Listing 11-10 shows the implementation of Powell's algorithm in Pharo. The following code example shows how to find the maximum of a vector function.

```
| fBlock educatedGuess hillClimber result |

fBlock := [ :v |
| x y |
x := v at: 1.
y := v at: 2.
((2 - x) * (2 - x)) negated + ((2 - y) * (2 - y)) negated ].
educatedGuess := #(100 100) asPMVector.
hillClimber := PMHillClimbingOptimizer maximizingFunction: fBlock.
hillClimber initialValue: educatedGuess.
result := hillClimber evaluate.
```

The class PMHillClimbingOptimizer is a subclass of class PMFunctionOptimizer. It has only one additional instance variable, unidimensionalFinder, to hold the one-dimensional optimizer used to find an optimum of the goal function along a given direction.

The method evaluateIteration uses the method `inject:into:` to perform steps 2-4 of the algorithm. Similarly step 5 of the algorithm is performed with a

method inject:into: within the method shiftDirection. This mode of using the iterator method inject:into: — performing an action involving two consecutive elements of an indexed collection — is somewhat unusual, but highly convenient[**?**]. The method minimizeDirection: implements step 3 of the algorithm.

Listing 11-10 Smalltalk implementation of Powell's algorithm

```
PMFunctionOptimizer subclass: #PMHillClimbingOptimizer
   instanceVariableNames: 'unidimensionalFinder'
   classVariableNames: ''
   package: 'Math-Numerical-Math-FunctionIterator'
```

```
PMFunctionOptimizer>>computeInitialValues
    unidimensionalFinder := PMOneVariableFunctionOptimizer
                                                    forOptimizer:
    self.
    unidimensionalFinder desiredPrecision: desiredPrecision.
    bestPoints := (1 to: result size)
                collect: [ :n | ( PMVectorProjectedFunction
                                     function: functionBlock)
                                   direction: ((PMVector
                                                   new: result
    size)
                                                       atAllPut: 0;
                                                       at: n put:
    1;
                                                       yourself);
                                            yourself
                        ]
```

```
PMFunctionOptimizer>>evaluateIteration
    | oldResult |
    precision := 1.
    bestPoints inject: result
                into: [ :prev :each | ( self minimizeDirection: each
                                                       from:
    prev)].
    self shiftDirections.
    self minimizeDirection: bestPoints last.
    oldResult := result.
    result := bestPoints last origin.
    precision := 0.
    result with: oldResult do:
        [ :x0 :x1 |
          precision := ( self precisionOf: (x0 - x1) abs relativeTo:
                                              x0 abs) max:
    precision.
        ].
    ^ precision
```

```
PMFunctionOptimizer>>minimizeDirection: aVectorFunction
    ^ unidimensionalFinder
        reset;
        setFunction: aVectorFunction;
        addPointAt: 0;
        addPointAt: precision;
        addPointAt: precision negated;
        evaluate
```

```
PMFunctionOptimizer>>minimizeDirection: aVectorFunction from: aVector
    ^aVectorFunction
        origin: aVector;
        argumentWith: (self minimizeDirection: aVectorFunction)
```

```
PMFunctionOptimizer>>shiftDirections
    | position delta firstDirection |
    firstDirection := bestPoints first direction.
    bestPoints inject: nil
                    into: [ :prev :each |
                            position isNil
                                ifTrue: [ position := each origin]
                                ifFalse:[ prev direction: each

    direction].
                            each
                            ].
    position := bestPoints last origin - position.
    delta := position norm.
    delta > desiredPrecision
        ifTrue: [ bestPoints last direction: (position scaleBy: (1 /

    delta))]
        ifFalse:[ bestPoints last direction: firstDirection]
```

11.7 Simplex algorithm

The simplex algorithm, invented by Nelder and Mead, provides an efficient way to find a good approximation of the optimum of a function starting from any place [**?**]. The only trap into which the simplex algorithm can run into is a local optimum. On the other hand, this algorithm does not converge very well in the vicinity of the optimum. Thus, it must not be used with the desired precision set to a very low value. Once the optimum has been found with the simplex algorithm, other more precise algorithms can then be used, such as the ones describes in section 11.1 or 11.6. MINUIT uses a combination of simplex and Newton algorithms. Our implementation of general purpose optimizer uses a combination of simplex and Powell algorithms.

A simplex in a n-dimensional space is a figure formed with $n + 1$ summits. For example, a simplex in a 2-dimensional space is a triangle; a simplex in a

3-dimensional space is a tetrahedron. Let us now discuss the algorithm for finding the optimum of a function with a simplex.

1. Pick up $n + 1$ points in the search space and evaluate the goal function at each of them. Let **A** be the summit yielding the worst function's value.
2. if the size of the simplex is smaller than the desired precision, terminate the algorithm.
3. Calculate **G**, the center of gravity of the n best points, that is all points except **A**.
4. Calculate the location of the symmetric point of **A** relative to **G**: $\mathbf{A}' = 2\mathbf{G} - \mathbf{A}$.
5. If $f(\mathbf{A}')$ is not the best value found so far go to step 9.
6. Calculate the point $\mathbf{A}'' = 2\mathbf{A}' - \mathbf{G}$, that is a point twice as far from **G** as $\mathbf{A}'$.
7. If $f(\mathbf{A}'')$ is a better value than $f(\mathbf{A}')$ build a new simplex with the point **A** replaced by the point $\mathbf{A}''$ and go to step 2.
8. Otherwise, build a new simplex with the point **A** replaced by the point $\mathbf{A}'$ and go to step 2.
9. Calculate the point $\mathbf{B} = \frac{(\mathbf{G} + \mathbf{A})}{2}$.
10. If $f(\mathbf{B})$ yields the best value found so far build a new simplex with the point **A** replaced by the point $\mathbf{A}''$ and go to step 2.
11. Otherwise build a new simplex obtained by dividing all edges leading to the point yielding the best value by 2 and go back to step 2.

Figure 11-11 shows the meaning of the operations involved in the algorithm in the 3 dimensional case. Step 6 makes the simplex grow into the direction where the function is the best so far. Thus, the simplex becomes elongated in the expected direction of the optimum. Because of its geometrical shape, the next step is necessarily taken along another direction, causing an exploration of the regions surrounding the growth obtained at the preceding step. Over the iterations, the shape of the simplex adapts itself to narrow valleys where the hill climbing algorithms notoriously get into trouble. Steps 9 and 11 ensures the convergence of the algorithm when the optimum lies inside the simplex. In this mode the simplex works very much like the golden section search or the bisection algorithms.

Finding the initial points can be done in several ways. If a good approximation of the region where the maximum might be located can be obtained one uses that approximation as a start and generate n other points by finding the optimum of the function along each axis. Otherwise, one can generate random points and select $n + 1$ points yielding the best values to build the initial simplex. In all cases, one must make sure that the initial simplex has a non-vanishing size in all dimensions of the space. Otherwise the algorithm will not reach the optimum.

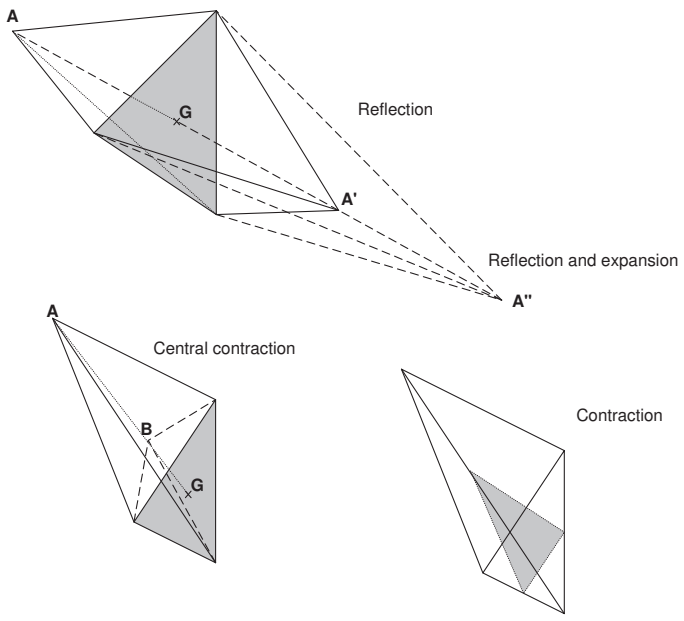


Figure 11-11 Operations of the simplex algorithm

Simplex algorithm

The class implementing the simplex algorithm belong to the hierarchy of the iterative processes discussed in chapter 4. The method evaluateIteration directly implements the steps of the algorithm as described above. The points **G**, $\mathbf{A}'$, $\mathbf{A}''$ and **B** are calculated using the vector operations described in section 8.1.

The routine initializeIterations assumes that an initial value has been provided. It then finds the location of an optimum of the goal function along each axis of the reference system starting each time from the supplied initial value, unlike hill climbing algorithms. Restarting from the initial value is necessary to avoid creating a simplex with a zero volume. Such mishaps can arise when the initial value is located on an axis of symmetry of the goal function. This can happen quite frequently with *e*ducated guesses.

Listing **??** shows the Smalltalk implementation of the simplex algorithm. The following code example shows how to invoke the class to find the minimum of a vector function.

```
| fBlock educatedGuess simplex result |

fBlock :=<the goal function>
educatedGuess :=<a vector in the search space>
```

```
simplex := PMSimplexOptimizer minimizingFunction: fBlock.
simplex initialValue: educatedGuess.
result := simplex evaluate
```

Except for the line creating the instance of the simplex optimizer, this code example is identical to the example of Powell's hill climbing algorithm (code example 11-9).

The class PMSimplexOptimizer is a subclass of class PMFunctionOptimizer. In order to be able to use the iterator methods efficiently, the worst point of the simplex, **A**, is held in a separate instance variable worstPoint. As we do not need to know the function's value $f(\mathbf{A})$, it is kept as a vector. The remaining points of the simplex are kept in the instance variable bestPoints of the superclass. Since this collection is sorted automatically when points are inserted to it, there is no explicit sorting step.

Listing 11-12 m

```
alltalk implementation of simplex algorithm
\label{ls:optimizersimplex}
\input{Smalltalk/Minimization/DhbSimplexOptimizer}
```

11.8 Genetic algorithm

All optimizing algorithm discussed so far have one common flaw: they all terminate when a local optimum is encountered. In most problems, however, one wants to find the absolute optimum of the function. This is especially true if the goal function represents some economical merit.

One academic example is the maximization of the function

$$f(\mathbf{x}) = \frac{\sin^2 |\mathbf{x}|}{|\mathbf{x}|^2}. \tag{11.7}$$

This function has an absolute maximum at $\mathbf{x} = 0$, but all algorithms discussed so far will end up inside a ring corresponding to $|\mathbf{x}| = n\pi/2$ where n is any positive odd integer.

In 1975 John Holland introduced a new type of algorithm — dubbed genetic algorithm — because it tries to mimic the evolutionary process identified as the cause for the diversity of living species by Charles Darwin. In a genetic algorithm the elements of the search space are considered as the chromosomes of individuals; the goal function is considered as the measure of the fitness of the individual to adapt itself to its environment[**?**][**?**]. The iterations are aping (pun intended) the Darwinian principle of survival and reproduction. At each iteration, the fittest individuals survive and reproduce themselves. To bring some variability to the algorithm mutation and crossover of chromosomes are taken into account.

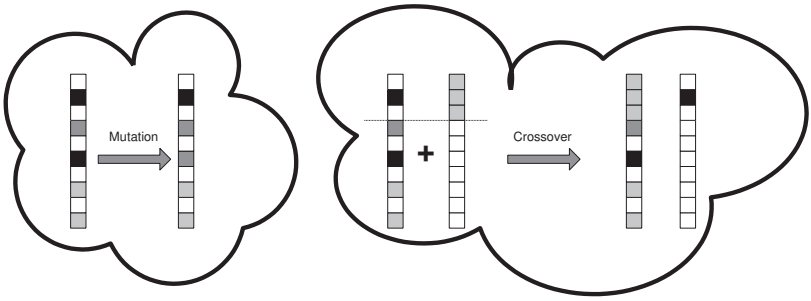


Figure 11-13 Mutation and crossover reproduction of chromosomes

Mutation occurs when one gene of a chromosome is altered at reproduction time. Crossover occurs when two chromosomes break themselves and recombine with the piece coming from the other chromosome. These processes are illustrated on figure 11-13. The point where the chromosomes are breaking is called the crossover point. Which individual survives and reproduces itself, when and where mutation occurs and when and where a crossover happens is determined randomly. This is precisely the random nature of the algorithm which gives it the ability to jump out of a local optimum to look further for the absolute optimum.

Mapping the search space on chromosomes

To be able to implement a genetic algorithm one must establish how to represent the genes of a chromosome. At the smallest level the genes could be the bits of the structure representing the chromosome. If the search space of the goal function do cover the domain generated by all possible permutations of the bits, this is a good approach. However, this is not always a practical solution since some bit combinations may be forbidden by the structure. For example, some of the combinations of a 64 bit word do not correspond to a valid floating point number.

In the case of the optimization of a vector function, the simplest choice is to take the components of the vector as the genes. Genetic algorithms are used quite often to adjust the parameters of a neural network [**?**]. In this case, the chromosomes are the coefficients of each neuron. Chromosomes can even be computer subprograms in the case of genetic programming [**?**]. In this latter case, each individual is a computer program trying to solve a given problem.

Figure 11-14 shows a flow diagram of a general genetic algorithm. The reproduction of the individual is taken literally: a copy of the reproducing individual is copied into the next generation. The important feature of a generic

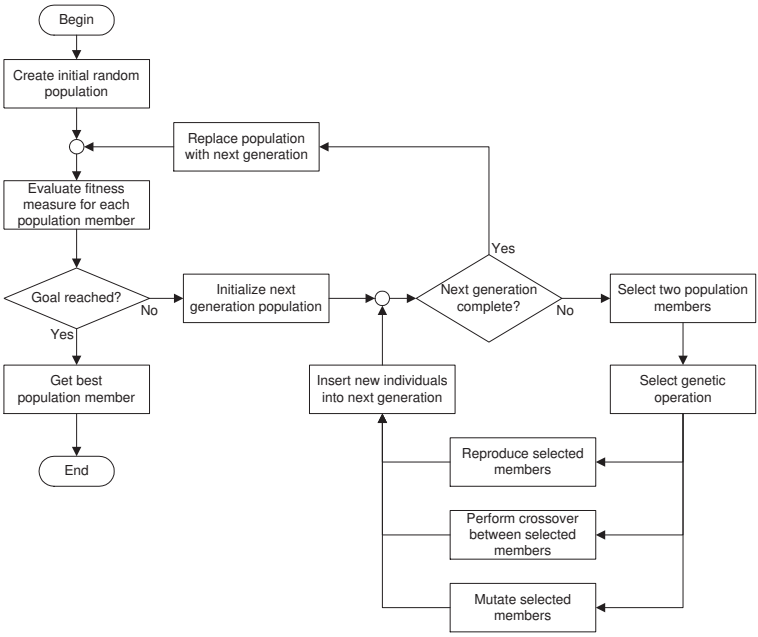


Figure 11-14 General purpose genetic algorithm

algorithm is that the building of the next generation is a random process. To ensure the survival of the fittest, the selection of the parents of the individuals of the next generation is performed at random with uneven probability: the fittest individuals have a larger probability of being selected than the others. Mutation enables the algorithm to create individuals having genes corresponding to unexplored regions of the search space. Most of the times such mutants will be discarded at the next iteration; but, in some cases, a mutation may uncover a better candidate. In the case of the function of equation 11.7, this would correspond to jumping from one ring to another ring closer to the function's maximum. Finally the crossover operation mixes good genes in the hope of building a better individual out of the properties of good inidividuals. Like mutation the crossover operation gives a stochastic behavior to the algorithm enabling it to explore uncharted regions of the search space.

> **Note:** Because of its stochastic nature a genetic algorithm is the algorithm of choice when the goal function is expressed on integers.

General implementation

The left hand side of the diagram of figure 11-14 is quite similar to the flow diagram of an iterative process (*c.f.* figure 4-2 in chapter 4). Thus, the class

implementing the genetic algorithm is a subclass of the iterative process class discussed in chapter 4.

The *g*enetic nature of the algorithm is located in the right hand side of the diagram of figure 11-14. As we have mentioned before the implementation of the chromosomes is highly problem dependent. All operations located in the top portion of the mentioned area can be expressed in generic terms without any knowledge of the chromosomic implementation. to handle the lower part of the right hand side of the diagram of figure 11-14, we shall implement a new object, the chromosome manager.

One should also notice that the value of the function is not needed when the next generation is build. Thus, the chromosome manager does not need to have any knowledge of the goal function. The goal function comes into play when transfering the next generation to the *m*ature population, that is, the population used for reproduction at the next iteration . At the maturity stage, the value of the goal function is needed to identify the fittest individuals. In our implementation, the next generation is maintained by the chromosome manager whereas the population of mature individuals is maintained by the object in charge of the genetic algorithm which has the knowledge of the goal function.

The chromosome manager has the following instance variables:

`populationSize` contains the size of the population; one should pick up a large enough number to be able to cover the search space efficiently: the larger the dimension of the space search space, the larger must be the population size;

`rateOfMutation` contains the probability of having a mutation while reproducing;

`rateOfCrossover` contains the probability of having a crossover while reproducing.

All of these variables have getter and setter accessor methods. In addition a convenience instance creation method is supplied to create a chromosome manager with given values for all three instance variables. The chromosome manager implements the following methods:

`isFullyPopulated` to signal that a sufficient number of individuals has been generated into the population;

`process` to process a pair of individuals; this method does the selection of the genetic operation and applies it; individuals are processed by pair to always have a possibility of crossover;

`randomnizePopulation` to generate a random population;

`reset` to create an empty population for the next generation.

Finally the chromosome manager must also implement methods performing each of the genetic operations: reproduction, mutation and crossover. The Smalltalk implementation supplies methods that returns a new individual.

The genetic optimizer is the object implementing the genetic algorithm proper. It is a subclass of the iterative process class described in chapter 4.2. In addition to the handling of the iterations the genetic optimizer implements the steps of the algorithm drawn on the top part of the right hand side of the diagram of figure 11-14. It has one instance variable containing the chromosome manager with which it will interact. The instance creation method take three arguments: the function to optimize, the optimizing strategy and the chromosome manager.

The method `initializeIteration` asks the chromosome manager to supply a random population. The method `evaluateIteration` performs the loop of the right hand side of the diagram of figure 11-14. It selects a pair of parents on which the chromosome manager performs the genetic operation.

Selecting the genetic operation is performed with a random generator. The values of the goal function are used as weights. Let $f(p_i)$ be the value of the goal function for individual p_i and let p_b and p_w be respectively the fittest and the lest fit individual found so far (b stands for best and w stands for worst). One first computes the unnormalized probability:

$$\tilde{P}_i = \frac{f(p_i) - f(p_w)}{f(p_b) - f(p_w)}. \tag{11.8}$$

This definition ensures that $\tilde{P}_i$ is always comprised between 0 and 1 for any goal function. Then we can use the discrete probability

$$P_i = \frac{1}{\sum \tilde{P}_i} \tilde{P}_i. \tag{11.9}$$

The sum in equation 11.9 is taken over the entire population. An attentive reader will notice than this definition assigns a zero probability of selecting the worst individuals. This gives a slight bias to our implementation compared to the original algorithm. This is can be easily compensated by taking a sufficiently large population. The method `randomScale` calculates the P_i of equation 11.9 and returns an array containing the integrated sums:

$$R_i = \sum_{k=0}^{i} P_i. \tag{11.10}$$

The array R_i is used to generate a random index to select individuals for reproduction.

The transfer between the next generation and the mature population is performed by the method `collectPoints`.

In the general case, there is no possibility to decide when the terminate the algorithm. In practice, it is possible that the population stays stable for quite a while until suddenly a new individual is found to be better than the rest. Therefore a criteria based on the stability of the first best points is likely to be beat the purpose of the algorithm, namely to jump out of a local optimum. Some problems can define a threshold at which the goal function is considered sufficiently good. In this case, the algorithm can be stopped as soon as the value of the goal function for the fittest individual becomes better than that threshold. In the general case, however, the implementation of the genetic algorithm simply returns a constant pseudo precision — set to one — and runs until the maximum number of iterations becomes exhausted.

Pharo implementation

Listing **??** shows the code of an abstract chromosome manager in Smalltalk and of a concrete implementation for vector chromosomes. The class PMChromosomeManager has one instance variable in addition to the variables listed in section 11.8: population. This variable is an instance of an OrderedCollection containing the individuals of the next generation being prepared.

The class PMVectorChromosomeManager is a subclass of class PMChromosomeManager implementing vector chromosomes. It has two instance variables

origin a vector containing the minimum possible values of the generated vectors;

range a vector containing the range of the generated vectors.

In other words origin and range are delimiting an hypercube defining the search space.

Listing 11-15 m

```
alltalk chromosome: abstract and concrete \label{ls:chromosome}
\input{Smalltalk/Minimization/DhbChromosomeManager}
\input{Smalltalk/Minimization/DhbVectorChromosomeManager}
```

Listing **??** shows how the genetic optimizer is implemented in Smalltalk. The following code example shows how to use a genetic optimizer to find the maximum of a vector function.

```
    | fBlock optimizer manager origin range result |
```

fBlock :=<the goal function>
origin :=<a vector containing the minimum expected value of the component>
range :=<a vector containing the expected range of the component>

```
    optimizer := PMGeneticOptimizer maximizingFunction: fBlock.
    manager := PMVectorChromosomeManager new: 100 mutation: 0.1
     crossover: 0.1.
    manager origin: origin; range: range.
    optimizer chromosomeManager: manager.
    result := optimizer evaluate
```

After establishing the goal function and the search space, an instance of the genetic optimizer is created. The next line creates an instance of a vector chromosome manager for a population of 100 individuals (sufficient for a 2-3 dimensional space) and rates of mutation and crossover equal to 10%. The next line defines the search space into the chromosome manager. The final line performs the genetic search and returns the result.

In Smalltalk the population of the next generation is maintained in the instance variable population. Each time a next generation has been established, it is transferred into a collection of best points by the method collectPoints. Each element of the collection bestPoints is an instance of an subclass of OptimizingPoint. The exact type of the class is determined by the search strategy. Since best points are sorted automatically, the result is always the position of the first element of bestPoints.

Listing 11-16 m

```
alltalk implementation of genetic algorithm
    \label{ls:optimizerabsgen}
\input{Smalltalk/Minimization/DhbGeneticOptimizer}
```

11.9 Multiple strategy approach

As we have seen most of the optimizing algorithms described so far have some limitation:

- Hill climbing algorithms may get into trouble far from the optimum and may get caught into a local optimum. This is exemplified in figure 11-17.
- The simplex algorithm may get caught into a local optimum and does not converge well near the optimum.
- Genetic algorithms do not have a clear convergence criteria.

After reading the above summary of the pro and cons of each algorithm, the reader may have already come to the conclusion that mixing the three algorithms together can make a very efficient strategy to find the optimum of a wide variety of functions.

One can start with a genetic optimizer for a sufficient number of iterations. This should ensure that the best points found at the end of the search does not lie too far from the absolute optimum. Then, one can use the simplex

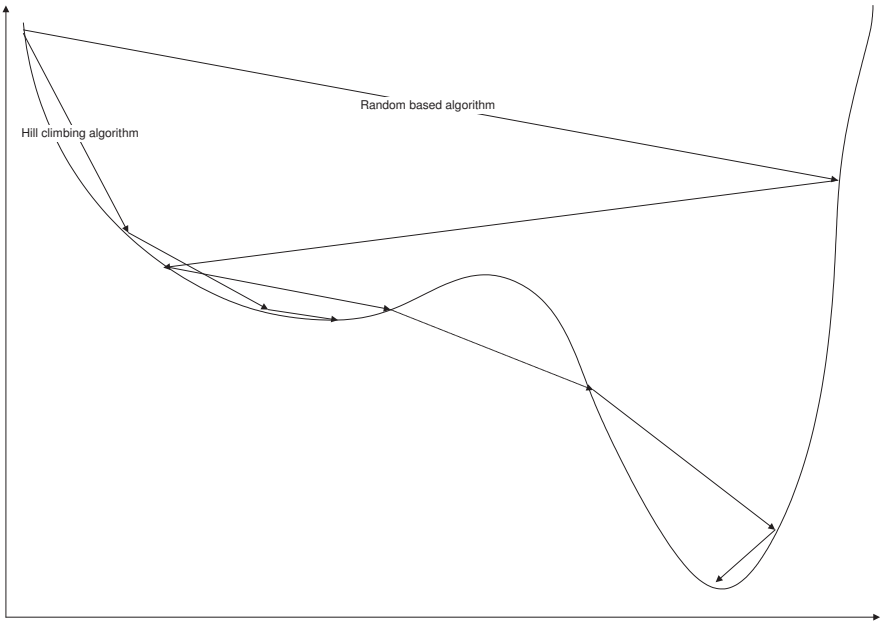


Figure 11-17 Compared behavior of hill climbing and random based algorithms.

algorithm to get rapidly near the optimum. The final location of the optimum is obtained using a hill climbing optimizer.

General implementation

This multiple strategy approach, inspired from the program MINUIT, has been adapted to the use of the algorithms discussed here. The class `MultiVariableGeneralOptim` combines the three algorithms: genetic, simplex and hill climbing, in this order. We could have make it a subclass of `Object`, but we decided to reuse all the management provided by the abstract optimizer class discussed in section 11.3. Therefore, our general purpose optimizer is a subclass of the abstract optimizer class although it does not really uses the framework of an iterative process. We only need one additional instance variable: the range used to construct the hypercube search space for the vector genetic chromosome manager. A corresponding setting method is provided: `setRange`.

The method `initializeIterations` performs search using the genetic algorithm as an option and, then, the simplex algorithm. Since the genetic algorithm require a great deal of function evaluate — due to its stochastic nature — it is a good idea to give the user the choice of by-passing the use of the genetic algorithm. If no range has been defined, only the simplex algorithm is used from the supplied initial value. Otherwise a search is made with the genetic algorithm using the initial value and the range to define the search space.

Then the simplex algorithm is started from the best point found by the genetic algorithm. The precision for the simplex search is set to the square root of the precision for the final search. Less precision is required for this step because the final search will give a better precision.

The method evaluateIteration performs the hill climbing algorithm and returns it precision. As the desired precision of the hill climbing algorithm is set to that of the general purpose optimizer. As a consequence, there will only be a single iteration.

Listing **??** shows the implementation in Smalltalk. At this point we shall abstain from commenting the code as the reader should have no more need for such thing. . . Hopefully!

Listing 11-18 m

```
alltalk
implementation of a general optimizer \label{ls:optimizergeneral}
\input{Smalltalk/Minimization/DhbMultiVariableGeneralOptimizer}
```

CHAPTER 12

Data mining

Creusez, fouillez, bêchez, ne laissez nulle place
Où la main ne passe et repasse,[1]
Jean de La Fontaine

Data mining is a catchy buzz-word of recent introduction covering activities formerly known as data analysis. The problem is akin to what we already have seen in chapter 10. In the case of data mining, however, the emphasis is put on large data sets. *Large* must be understood in to ways: first, each *data point* is actually made of a large number of measurements; second, the number of data points is large, even huge. The expression data mining was coined when large corporate databases become common place. The original reason for the presence of these databases was the day to day dealing with the business events. A few people started realizing that these databases are containing huge amount of information, mostly of statistical nature, about the type of business. That information was just waiting to be mined just like the Mother Lode waited for the coming of the 49ers. Hence the term data mining.

Figure 12-1 shows the classes described in this chapter. There are two aspects to the data mining activity: one is preparing the data in order to render them suitable for the processing; the second consists of extracting the information, that is, the processing of the data. Depending on the problems and on the technique used during the second step , the first step is not always needed.

Figure 12-1 Classes used in data mining

[1]Dig, search, excavate, do not leave a place where your hands did not go once or more.

In any case, the preparation of the data is usually very specific to the type of data to be analyzed. Therefore one can only make very general statements about this problem: one must watch for rounding errors; one must adjust the scale of data having widely different ranges; one must check the statistical validity of the data sample; *etc*. We shall say no more about this first aspect of data mining, but we wanted to warn the reader about this important issue.

Finally, data mining differs from estimation in that, in many cases the type of information to be extracted is not really known in advance. Data mining tries to identify trends, common properties and correlation between the data. The goal of data mining is usually to reduce large sample to much smaller manageable sets, which can be efficiently targeted. One example, is the selection of a sample of customers suitable to respond to a new product offered by a bank or an insurance company. Mailing operations are costly; thus, any reduction of the mailing target set with a significant improvement of the probability of response can bring a significant saving. Another example in the medical domain is the scanning for certain type of cancer, which are expensive[2]. If one can identify a population with high risk of cancer, the scanning only needs to done for that population, thus keeping the cost low.

A good collection of articles about the techniques exposed in this chapter can be found in [**?**].

12.1 Data server

As we have said in the introduction, data mining means handling large amounts of data, most likely more than the computer memory can hold. Thus, we need an object to handle these data for all objects implementing data mining techniques.

Figure 12-1 with the boxes **AbstractDataServer** and **MemoryBasedDataServer** grayed.

The data server object needs to implement five functionalities:

1. opening the physical support of the data,
2. getting the next data item,
3. checking whether more data items are available,
4. repositioning the data stream at its beginning and
5. closing the physical support of the data.

Depending on the problem at hand, the responsibility of a data server can extend beyond the simple task of handling data. In particular, it could be charged of performing the data preparation step mentioned in the introduction. In our implementation, we give two classes. One is an abstract class from which all data servers used by the data mining objects described in this chapter must derive. The data item returned by the method returning the

[2] In medical domain, *expensive* is not necessarily a matter of money. It can mean high risk for the patient or the examination is regarded as too intrusive by the patient.

nest item is a vector object whose components are the data corresponding to one measurement. The second class of data server is a concrete class implementing the data server on a collection or array kept in the computer's memory. Such server is handy for making tests.

> **Note:** Examples of use of data server are given in the other sections; no code example are given here.

Data server — Smalltalk implementation

Listing **??** shows the implementation of the abstract data server in Smalltalk. The implementation of the concrete class is shown in listing **??**.

Our implementation uses the same methods used by the hierarchy of the class `Stream`.

Listing 12-2 m

```
alltalk abstract data server \label{ls:abstractserver}
\input{Smalltalk/DataMining/DhbAbstractDataServer}
```

Listing 12-3 m

```
alltalk memory based data server \label{ls:memoryserver}
\input{Smalltalk/DataMining/DhbMemoryBasedDataServer}
```

12.2 Covariance and covariance matrix

When one deals with two or more random variables an important question to ask is whether or not the two variables are dependent from each other.

For example, if one collects the prices of homes and the incomes of the home owners,one will find that inexpensive homes are mostly owned by low income families, *mostly but not always*. It is said that the price of a home is *correlated* with the income of the home owner. As soon as one deals with more than two variables things stop being clear cut. Correlations become hard to identify especially because of these *mostly but not always* cases. Therefore, one must find a way to expressed mathematically how much two random variables are correlated.

Let $x_1, \ldots, x_m$ be several random variable. They can be considered as the components of a m-dimensional(random) vector $\mathbf{x}$. The probability density function of the vector $\mathbf{x}$ is denoted $P(\mathbf{x})$; it measures the probability of observing a vector within the differential volume element located at $\mathbf{x}$. The average of the vector $\mathbf{x}$ is defined in a way similar to the case of a single random variable. The i component of the average is defined by

$$\mu_i = \int \ldots \int x_i P(\mathbf{x})\, dx_1 \ldots dx_m. \tag{12.1}$$

The covariance matrix of the random vector $\mathbf{x}$ gives a measure of the correlations between the components of the vector $\mathbf{x}$. The components of the covariance matrix are defined by

$$\varrho_{ij} = \int \dots \int (x_i - \mu_i)(x_j - \mu_j) P(\mathbf{x}) \, dx_1 \dots dx_m. \tag{12.2}$$

As one can see the covariance matrix is a symmetric matrix. It is also positive definite. Furthermore ϱ_{ii} is the variance of the i component of the random vector.

> **Note:** The error matrix of a least square or maximum likelihood fit — discussed in chapter 10 — is the covariance matrix of the fit parameters.

If two components are independent, their covariance — that is, the corresponding element of the covariance matrix — is zero. The inverse is not true, however. For example, consider a 2-dimensional vector with components (zz^2) where z is a random variable. If z is distributed according to a symmetric distribution, the covariance between the two components of the vector is zero. Yet, the components are 100% dependent from each other by construction.

The correlation coefficient between components i and j of the vector $\mathbf{x}$ is then defined by

$$\rho_{ij} = \frac{\varrho_{ij}}{\sigma_i \sigma_j}, \tag{12.3}$$

where $\sigma_i = \sqrt{\varrho_{ii}}$ is the standard deviation of the i component of the vector $\mathbf{x}$. By definition, the correlation coefficient is comprised between -1 and 1. If the absolute value of a correlation coefficient is close to 1 then one can assert that the two corresponding components are indeed correlated.

If the random vector is determined experimentally, one calculates the estimated covariance with the following statistics

$$\operatorname{cov}(x_i, x_j) = \frac{1}{n} \sum_{k=1}^{n} (x_{i,k} - \mu_i)(x_{j,k} - \mu_j), \tag{12.4}$$

where $x_{i,k}$ is the i component of the k measurement of the vector $\mathbf{x}$. Similarly the estimated correlation coefficient of the corresponding components is defined as

$$\operatorname{cor}(x_i, x_j) = \frac{\operatorname{cov}(x_i, x_j)}{s_i s_j}, \tag{12.5}$$

where s_i is the estimated standard deviation of the i component.

Like for the central moment, there is a way to compute the components of the covariance matrix while they are accumulated. If $\operatorname{cov}_n(x_i, x_j)$ denotes the estimated covariance over n measurements, one has:

$$\operatorname{cov}_{n+1}(x_i, x_j) = \frac{n}{n+1} \operatorname{cov}_n(x_i, x_j) + n \Delta_{i,n+1} \Delta_{j,n+1}, \tag{12.6}$$

where $\Delta_{x,n+1}$ and $\Delta_{y,n+1}$ are the corrections to the averages of each variable defined in equation 9.12 of section 9.2. The derivation of equation 12.6 is given in appendix **??**.

Using covariance information

A covariance matrix contains statistical information about the set of measurements over which is has been determined. There are several ways of using this information.

The first approach uses the covariance matrix directly. The best example is the analysis known as the shopping cart analysis [**?**]. For example, one can observe that consumers buying cereals are buying low fat milk. This can give useful information on how to target special sales efficiently. Application working in this mode can use the code of sections **??** or **??** as is.

Another approach is to use the statistical information contained in a covariance matrix to process data coming from measurements which were not used to determine the covariance matrix. In the rest of this chapter we shall call the set of measurements, which is used to determine the covariance matrix, the *calibrating set*. In this second mode of using a covariance matrix, measurements are *compared* or *evaluated* against those of the calibrating set. It is clear that the quality of the information contained in the covariance matrix depends on the quality of the calibrating set. We shall assume that this is always the case. Techniques working according to this second mode are described in sections **??**, **??** and **??**.

Covariance matrix — General implementation

Figure 12-1 with the boxes **VectorAccumulator** and **CovarianceAccumulator** grayed.

The object in charge of computing the covariance matrix of a series of measurements is implemented as for central moments. Because we shall need to only compute the average of a vector, the implementation is spread over two classes, one being the subclass of the other for efficient reuse.

The class `VectorAccumulator` has two instance variables:

`count` counts the number of vectors accumulated in the object so far;

`average` keeps the average of the accumulated vector;

The class `CovarianceAccumulator` is a subclass of the class `VectorAccumulator`. It has one additional instance variable:

`covariance` accumulates the components of the covariance matrix; for efficiency reason, only the lower half of the matrix is computed since it is symmetric.

The topmost class implements equation 9.12 in the method `accumulate`. The subclass overloads this method to implement equation 12.6.

Covariance matrix — Smalltalk implementation

Listing **??** shows the implementation of the accumulation of a vector in Smalltalk. Listing **??** shows the implementation of the accumulation of the covariance matrix. The following code example shows how to accumulate the average of a series of vectors read from a data stream.

```
| accumulator valueStream average |
accumulator := PMVectorAccumulator new.
valueStream open.
[ valueStream atEnd ]
       whileFalse: [ accumulator accumulate: valueStream next ].
valueStream close.
average := accumulator average.
```

The reader can see that this example is totally equivalent to the code example **??**. Here the method `next` of the data stream must return a vector instead of a number; all vectors must have the same dimension. The returned average is a vector of the same dimension.

The next code example shows how to accumulate the both average and covariance matrix. The little differences with the preceding example should be self explanatory to the reader.

```
| accumulator valueStream average covarianceMatrix |
accumulator := DhbCovarianceAccumulator new.
valueStream open.
[ valueStream atEnd ]
       whileFalse:[ accumulator accumulate: valueStream next ].
valueStream close.
average := accumulator average.
covarianceMatrix := accumulator covarianceMatrix.
```

The method `accumulate` of class `DhbVectorAccumulator` answers the corrections to each component of the average vector. This allows the class `PMCovarianceAccumulator` to reuse the results of this method. In class `PMVectorAccumulator`, vector operations are used. The method `accumulate` of class `PMCovarianceAccumulator` works with indices because one only computes the lower half of the matrix.

Listing 12-6 m

```
alltalk implementation of vector average \label{ls:vectoraverage}
\input{Smalltalk/DataMining/DhbVectorAccumulator}
```

Listing 12-7 m

```
alltalk implementation of covariance matrix \label{ls:covmatrix}
\input{Smalltalk/DataMining/DhbCovarianceAccumulator}
```

12.3 Multidimensional probability distribution

To get a feeling of what the covariance matrix represents, let us now consider a vector **y** whose components are independent random variables distributed according to a normal distribution. The probability density function is given by

$$P(\mathbf{y}) = \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi\sigma_i^2}} e^{-\frac{\left(y_i-\mu_i\right)^2}{2\sigma_i^2}}. \tag{12.7}$$

In this case, the covariance matrix of the vector **y** is a diagonal matrix $\tilde{\mathbf{V}}$, whose diagonal elements are the variance of the vector's components. If $\tilde{\mathbf{C}}$ is the inverse of the matrix $\tilde{\mathbf{V}}$, equation **??** can be rewritten as

$$P(\mathbf{y}) = \sqrt{\frac{\det \tilde{\mathbf{C}}}{(2\pi)^m}} e^{-\frac{1}{2}(\mathbf{y}-\bar{\mathbf{y}})^{\mathrm{T}} \tilde{\mathbf{C}}\,(\mathbf{y}-\bar{\mathbf{y}})}, \tag{12.8}$$

where $\bar{\mathbf{y}}$ is the vector whose components are $\mu_1, \ldots, \mu_m$. Let us now consider a change of coordinates $\mathbf{x} = \mathbf{O}\mathbf{y}$, where **O** is an orthogonal matrix. We have already met such transformations in section 8.7. Because the matrix is orthogonal, the differential volume element is invariant under the change of coordinates. Thus, the probability density function of the vector **x** is

$$P(x) = \sqrt{\frac{\det C}{(2\pi)^m}} e^{-\frac{1}{2}(x-\bar{x})^T C (x-\bar{x})}, \tag{12.9}$$

where the matrix **C** is equal to $\mathbf{O}^{\mathrm{T}}\tilde{\mathbf{C}}\mathbf{O}$. The vector $\bar{\mathbf{x}}$ is simply[3] equal to $\mathbf{O}\bar{\mathbf{y}}$. The covariance matrix, **V**, of the vector **x** is then equal to $\mathbf{O}^{\mathrm{T}}\tilde{\mathbf{V}}\mathbf{O}$. It is also the inverse of the matrix **C**. Thus, equation **??** can be rewritten as

$$P(x) = \frac{1}{\sqrt{(2\pi)^m \det V}} e^{-\frac{1}{2}(x-\bar{x})^T \mathbf{V}^{-1} (x-\bar{x})}. \tag{12.10}$$

In the case of a normal distribution in a multi-dimensional space, the covariance matrix plays the same role as the variance in one dimension.

12.4 Covariance data reduction

Reversing the derivation of the preceding section one can see that the eigenvalues of the covariance matrix of the vector **x** correspond to the variances of a series of independent[4] random variables $y_1, \ldots, y_m$. Since the covariance matrix is symmetric these eigenvalues as well as the matrix **O** describing the change of coordinates can be obtained using Jacobi's algorithm described in section 8.7.

[3] All this is a consequence of the linear property of the expectation value operator.
[4] Not necessarily normally distributed!

If some eigenvalues are much larger than the others, one can state that the information brought by the corresponding variables brings little information to the problem. Thus, one can omit the corresponding variable from the rest of the analysis.

Let $\sigma_1^2, \ldots, \sigma_m^2$ be the eigenvalues of the covariance matrix such that $\sigma_i^2 < \sigma_j^2$ for $i < j$. Let us assume that there exist an index k such that $\sigma_k^2 \gg \sigma_{k-1}^2$. Then, The rest of the data analysis can be made with a vector with components $y_k, \ldots, y_m$ where the vector $\mathbf{y}$ is defined by $\mathbf{y} = \mathbf{o}^{\mathsf{T}} x$.

This reduction technique has been used successfully in high energy physics[5] under the name principal component analysis. The data reduction allows to extract the relevant parameters of a complex particle detector to facilitate the quick extraction of the physical data — momentum and energy of the particle — from the observed data.

This kind of data reduction can be implemented within the framework of the data server described in section 12.1. The concrete implementation of such a server is straight forward. All needed objects — covariance accumulation, eigenvalues of a symmetric matrix, vector manipulation — have been discussed in different chapters. The rest of the implementation is specific to the problem at hand and can therefore not be treated on a general basis.

12.5 Mahalanobis distance

Mahalanobis, an Indian statistician, introduced this distance in the 30's when working on anthropometric statistics. A paper by Mahalanobis himself can be found in [**?**]. Readers interested by the historical dimension of the Mahalanobis distance can consult the paper by Das Gupta[6].

By definition, the exponent of equation **??** is distributed according to a χ^2 distribution with m degrees of freedom. This exponent remains invariant under the change of coordinates discussed in section **??**. Thus, the exponent of equation **??** is also distributed according to a χ^2 distribution with m degrees of freedom, where m is the dimension of the vector x. The Mahalanobis distance, d_M, is defined as the square root of the exponent, up to a factor $\sqrt{2}$. We have

$$d_M^2 = (\mathbf{x} - \bar{\mathbf{x}})^T \mathbf{V}^{-1} (\mathbf{x} - \bar{\mathbf{x}}) . \tag{12.11}$$

The Mahalanobis distance is a distance using the inverse of the covariance matrix as the metric. It is a distance in the geometrical sense because the covariance matrix as well as its inverse are positive definite matrices. The

[5] H.Wind, *Pattern recognition by principal component analysis of border regions*, Proceedings 3 th topical meeting on multi-dimensional analysis of high energy data, Nijmegen, 8-11 March 1978, W. Kittel, University of Nijmegen, 1978, pp. 99-106.

[6] Somesh Das Gupta, *The evolution of the D^2-statistics of Mahalanobis, Indian J. Pure Appl. Math., 26(1995), no. 6, 485-501.*

metric defined by the covariance matrix provides a normalization of the data relative to their spread.

The Mahalanobis distance — or its square — can be used to measure how *close* an object is from another when these objects can be characterized by a series of numerical measurements. Using the Mahalanobis distance is done as follows

1. the covariance matrix of the measured quantities, $\mathbf{V}$, is determined over a calibrating set;
2. one compute the inverse of the covariance matrix, $\mathbf{V}^{-1}$;
3. the distance of a new object to the calibrating set is estimated using equation **??**; if the distance is smaller than a given threshold value, the new object is considered as belonging to the same

One interesting property of the Mahalanobis distance is that it is normalized. Thus, it is not necessary to normalize the data provided rounding errors in inverting the covariance matrix are kept under control. If the data are roughly distributed according to a normal distribution the threshold for accepting whether or not an object belongs to the calibrating set can be determined from the χ^2 distribution.

Examples of use

The Mahalanobis distance can be applied in all problems where measurements must be classified.

A good example is the detection of coins in a vending machine. When a coin is inserted into the machine, a series of sensors gives several measurements, between a handful and a dozen. The detector can be calibrated using a set of good coins forming a calibration set. The coin detector can differentiate good coins from the fake coins using the Mahalanobis distance computed on the covariance matrix of the calibration set. Figure **??** shows an example of such data[7]. The light gray histogram is the distribution of the Mahalanobis distance of the good coins; the dark gray histogram that of the fake coins. The dotted line is the χ^2-distribution corresponding to the degrees of freedom of the problem; in other words, the distribution was not fitted. The reader can see that the curve of the χ^2-distribution reproduces the distribution of the experimental data. Figure **??** also shows the power of separation achieved between good and fake coins.

Another field of application is the determination of cancer cells from a biopsy. Parameters of cells — size and darkness of nucleus, granulation of the membrane — can be measured automatically and expressed in numbers. The covariance matrix can be determined either using measurements of healthy

[7] Data are reproduced with permission; the owner of the data wants to remain anonymous, however.

Figure 12-8 Using the Mahalanobis distance to differentiate between good and fake coins.

cells or using measurements of malignant cells. Identification of cancerous cells can be automatized using the Mahalanobis distance.

Mahalanobis distance — General implementation

Figure 12-1 with the box **Mahalanobis-Center** grayed.

The final goal of the object implementing the Mahalanobis distance is to compute the square Mahalanobis distance as defined in equation **??**. This is achieved with the method `distanceTo`. The inverse of the covariance matrix as well as the average vector $\bar{\mathbf{x}}$ are contained within the object. We have called the object a Mahalanobis center since it describes the *center* of the calibrating set.

To have a self-contained object, the Mahalanobis center is acting as a Facade to the covariance accumulator of section 12.2 for the accumulation of measurements. The methods `accumulate` and `reset` are delegated to an instance variable holding a covariance accumulator.

The Mahalanobis center has the following variables

accumulator a covariance accumulator as described in section 12.2;

center the vector $\bar{\mathbf{x}}$ and

inverseCovariance the inverse of the covariance matrix, that is $\mathbf{V}^{-1}$.

Our implementation is dictated by its future reuse in cluster analysis (*c.f.* section **??**). There, we need to be able to accumulate measurements while using the result of a preceding accumulation. Thus, computation of the center and the inverse covariance matrix must be done explicitly with the method computeParameters.

There are two ways of creating a new instance. One is to specify the dimension of the vectors which will be accumulated into the object. The second supplies a vector as the tentative center. This mode is explained in section **??**.

Mahalanobis distance — Smalltalk implementation

Listing **??** shows the implementation of a Mahalanobis center in Smalltalk. The following code example shows how to sort measurements using the Mahalanobis distance.

Code example 12.1

```
| center calibrationServer dataServer data threshold|
center := DhbMahalanobisCenter new: 5.
```

<The variable calibrationServer is setup to read measurements from the calibrating set>

```
calibrationServer open.
[ calibrationServer atEnd ]
     whileFalse: [ center accumulate: calibrationServer next ].
calibrationServer close.
center computeParameters.
```

<The variable dataServer is setup to read the measurements to be sorted between accepted and rejected; the variable threshold must also be determined or given>

```
dataServer open.
[ dataServer atEnd ]
     whileFalse: [ data := dataServer next.
                   ( center distanceTo: data) > threshold
                       ifTrue: [ self reject: data ]
                       ifFalse:[ self accept: data ].
                 ].
dataServer close.
```

The first line after the declaration creates a new instance of a Mahalanobis center for vectors of dimension 5. After setting up the server for the calibrating set data from the calibrating set are accumulated into the Mahalanobis center. At the end the parameters of the Mahalanobis center are computed. Then, the server for the other measurements is set up. The loop calculates the distance to the Mahalanobis center for each measurements. Our example supposes that the object executing this code has implemented two methods `accept` and `reject` to process accepted and rejected data.

Listing 12-9 m

```
alltalk Mahalanobis center  \label{ls:mahalanobis}
\input{Smalltalk/DataMining/DhbMahalanobisCenter}
```

12.6 Cluster analysis

Cluster analysis — also known as K-cluster — is a method to identify similarities between data. If the dimension of the data is less than or equal than 3, graphical data representation provides an easy way to identify data points having some similarity. For more than 3 measurements, the human brain is unable to clearly identify clustering.

Cluster analysis have been used successfully by the US army to define a new classification of uniform sizes and in banks [**?**]. British Telecom has used cluster analysis to detect a phone fraud of large scale in the early 90's[8].

The K-cluster algorithm goes as follows [**?**]:

1. Pick up a set of centers where possible clusters may exist;

[8] Private communication to the author.

2. place each data point into a cluster corresponding to the nearest center;
3. when all data points have been processed, compute the center of each cluster;
4. if the centers have changed, go back to point 2.

This algorithm nicely maps itself to the framework of successive approximations discussed in section 4.1. We now will investigate the steps of the algorithm in details.

Algorithm details

Picking up a set of centers corresponds to the box labeled *Compute or choose initial object* of figure 4-3. Since one is looking for unknown structure in the data there is little chance to make a good guess on the starting values. The most general approach is to pick up a few points at random from the existing data points to be the initial cluster's centers.

The next two steps correspond to the box labeled *Compute next object* of figure 4-3. Here lies the gist of the K-cluster algorithm. For each data point one first finds the cluster whose center is the nearest to the data point. What is the meaning of near? It depends on the problem. Let us just say at this point that one needs to have a way of expressing the distance between two data points. For the algorithm to converge the distance must be a distance in the geometrical sense of the term. In geometry, a distance is a numerical function of two vectors, $d(x, \mathbf{y})$. For all vectors **x**, **y** and **z** the following conditions must be met by the function

$$\begin{aligned} &d(x,y) \geq 0, \\ &d(x,y) = d(y,x), \\ &d(x,y) \leq d(x,z) + d(z,y). \end{aligned} \tag{12.12}$$

Furthermore, $d(x,y) = 0$ if and only if $x = y$. The simplest known distance function is the Euclidean distance expressed as

$$d_E(x,y) = \sqrt{(x-y)\cdot(x-y)}. \tag{12.13}$$

The square root in equation **??** is required for the distance function to behave linearly on a one dimensional subspace. The Euclidean distance corresponds to the notion of distance in everyday life. In assessing the proximity of two points, the square root is not needed.

After all points have been processed and assigned to a cluster, the center of each cluster is obtained by taking the vector average of all data points belonging to that cluster.

Then, one needs to determine whether the clusters have changed since the last iteration. In the case of the K-cluster algorithm it is sufficient to count

Figure 12-10 Example of cluster algorithm

the number of data points changing clusters at each iteration. When the same data point are assigned to the same clusters at the end of an iteration, the algorithm is completed. The *precision* used to stop the iterative process is an integer in this case.

Figure **??** shows how the algorithm works for a 2-dimensional case. Data points were generated randomly centered around 3 separated clusters. Three random points were selected as starting points: in this example the random choice was particularly unlucky since the three starting points all belong to the same cluster. Nevertheless, convergence was attained after 5 iterations[9].

Fine points

This simple example is admittedly not representative. However, this is not because of the small dimension nor because of the well separated clusters. It is because we knew in advance the number of clusters, 3 in this case. If we had started with 5 initial clusters, we would have ended up with 5 clusters, according to the expression, garbage in, garbage out, well-known among programmers.

One can modify the original algorithm to prune insignificant clusters from the search. This additional step must be added between steps 2 and 3. How to characterize an insignificant cluster? This often depends on the problem at hand. One thing for sure is that clusters with 0 elements should be left out. Thus, one easy method to prune insignificant clusters is to place a limit of the number of data points contained in each cluster. That limit should not be too high, however, otherwise most clusters will never get a chance of accumulating a significant amount of data points during the first few iterations.

Cluster analysis — General implementation

Figure 12-1 with the boxes ClusterFinder and Cluster grayed.

Our implementation of the K-cluster algorithm uses two classes: ClusterFinder and Cluster.

The class Cluster describes the behavior of a cluster. It is an abstract class. Subclasses implements the various strategies needed by the algorithm: distance calculation and pruning of insignificant clusters. The abstract class has only one instance variable, previousSampleSize, keeping the count of data points accumulated in the previous iteration. A cluster must be able to return the number of data points contained in the cluster. The method changes gives the number of data points which have changed cluster since the last iteration. The method isInsignificantIn determines whether a cluster must be

[9]The last iteration is not visible since the centers did not change.

pruned from the process. The method isUndefined allows the identification of clusters whose centers have not yet been defined. This method is used by the cluster finder to initialize undefined clusters with the data points before starting the iterative process.

An instance of a subclass of cluster must implement the following methods to interact with the class ClusterFinder:

distanceTo the argument of this method is a vector representing the data point; the returned value is the distance between the supplied vector and the center of the cluster; any subclass of Cluster can implement a suitable distance function as well as its own representation of the center;

accumulate the argument of this method is a vector representing the data point; this method is called when the cluster finder has determined that the data point must be placed within this cluster;

changes this method returns the number of data points which have been added to and removed from the cluster since the last iteration; the default implementation only calculates the difference between the number currently in the cluster and the number in the previous iteration; this simple approach works in practice[10];

sampleSize this method returns the number of data points actually accumulated into the cluster;

reset this method calculates the position of the new center at the end of an iteration; the default implementation stores the size of the previous sample.

> **Note:** A reader knowledgeable in patterns will think of using a Strategy pattern to implement the distance. I have tried this approach, but the resulting classes were much more complex for little gain. It is easier to implement subclasses of Cluster. Each subclass can not only implement the distance function, but can also choose how to accumulate data points: accumulation or storage.

The concrete class EuclideanCluster calculates the square of the Euclidean distance as defined in equation **??**. Using Euclidean distance requires that the components of the data vector have comparable scales. Cluster analysis with Euclidean clusters requires data preparation in general.

The class ClusterFinder implements the algorithm itself. This class is a subclass of the class for iterative processes described in section 4.1. The result of the iterative process is an array of clusters. The class ClusterFinder needs the following instance variables:

[10] Of course, one can construct cases where this simple approach fails. Such cases, however, correspond to situations where data points are oscillating between clusters and, therefore, do not converge. I have never met such cases in practice.

dataServer a data server object as described in section 12.1; this object is used to iterate on the data points;

dataSetSize a counter to keep track of the number of data points; this instance variable is combined with the next to provide a first order pruning strategy;

minimumRelativeClusterSize the minimum relative size of a significant cluster; the minimum cluster size is computed at each iteration by taking the product of this variable with the variable dataSetSize.

The class ClusterFinder uses an instance of the data server to iterate over the data points. before starting the search the client application can assign the list of initial clusters (step 1 of the algorithm . By default, the minimum relative size of the cluster is set to 0. A convenience method allows creating instances for a given data server and an initial set of clusters.

The method initializeIterations scans all clusters and looks for undefined clusters. The center of each undefined cluster is assigned to a data point read from the data server. This assumes that the data points have been collected randomly.

The method evaluateIteration processes each data point: first, it finds the index of the cluster nearest to the data point; then, the data point is accumulated into that cluster. After processing the data points, the clusters are processed to compute the new position of their centers and insignificant clusters are removed from the search. These tasks are performed within a method named collectChangesAndResetClusters. This method returns the number of data points which have changed cluster. The determination whether or not a cluster is significant is delegated to each cluster with the method isInsignificant. Thus, any subclass of cluster can implement its own strategy. The argument of the method isInsignificant is the cluster finder to provide each cluster with global information if needed.

The method finalizeIterations just closes the data server.

Cluster analysis — Smalltalk implementation

Listing **??** shows the implementation of the K-cluster algorithm in Smalltalk. The following code example shows how to implement a search for clusters.

Code example 12.2

```
| dataServer finder clusters |
```

<The variable dataServer is setup to read measurements from the calibrating set>

```
finder := DhbClusterFinder new: 5 server: dataServer
```

type: <a concrete subclass of Cluster>.

```
finder minimumRelativeClusterSize: 0.1.
clusters := finder evaluate.
```

After setting up a data server to read the data point, an instance of class DhbClusterFinder is created. the number of desired clusters is set to 5. The class of the clusters is specified. The next line sets the minimum relative size of each cluster to be kept during the iteration. Finally, the K-cluster algorithm is performed and clusters are retrieved from the finder object.

The abstract class DhbCluster is implemented with an instance variable accumulator. The cluster delegates the responsibility of accumulating the data points to this variable. It is assumed that the object in accumulator implements the interface defined by the vector accumulators described in section **??**.

The class DhbClusterFinder can be created in two ways. An application can set the list of initial clusters and the data server using the methods cluster and dataServer respectively. The convenience class creation method new:server:type: allows to specify the initial number of clusters, the data server and the class of the clusters. When this method is used, the collection of clusters is created when the instance of DhbClusterFinder is initialized; each cluster is created in an undefined state.

Listing 12-11 m

```
alltalk $K$-cluster algorithm \label{ls:clusterfinder}
\input{Smalltalk/DataMining/DhbCluster}
\input{Smalltalk/DataMining/DhbClusterFinder}
```

Listing **??** shows the implementation of the concrete cluster class DhbEuclideanCluster. The corresponding accumulator is an instance of class DhbVectorAccumulator. Data points are directly accumulated into the accumulaotr; individual data points are not kept.

Listing 12-12 m

```
alltalk implementation of an Euclidean cluster
    \label{ls:euclidcluster}
\input{Smalltalk/DataMining/DhbEuclideanCluster}
```

12.7 Covariance clusters

As we have seen in section **??** the Mahalanobis distance is a distance in the geometrical sense. Thus, this distance can be used by the K-cluster algorithm. We call clusters using the Mahalanobis distance covariance clusters since the metric for the distance is based on the covariance matrix.

The normalizing properties of the Mahalanobis distance makes it ideal for this task. When Euclidean distance is used, the metric remains the same in all directions. Thus, the extent of each cluster has more or less circular shapes. With the Mahalanobis distance the covariance metric is unique for each cluster. Thus, covariance clusters can have different shapes since the

metric adapts itself to the shape of each cluster. As the algorithm progresses the metric changes dynamically.

Figure 12-1 with the boxes CovarianceCluster grayed.

Covariance clusters need little implementation. All tasks are delegated to a Mahalanobis center described in section **??**. Listing **??** shows the Smalltalk implementation.

Listing 12-13 m

```
-alltalk covariance cluster \label{ls:mahacluster}
\input{Smalltalk/DataMining/DhbCovarianceCluster}
```